GESTOR DE CONTENIDOS
PARA ADMINISTRACIÓN DE WEB CORPORATIVA

UNIVERSIDAD
COMPLUTENSE
MADRID

Autor: Carlos Arroyo Aguilera
Tutor: Manuel Montenegro Montes
Curso académico: 2018/19
Trabajo de fin de grado del Grado en Ingeniería del Software
Facultad de Informática
Universidad Complutense de Madrid

ÍNDICE

AGRADECIMIENTOS

Quiero agradecer en primer lugar a Manuel Montenegro, director del TFG, todo su tiempo y dedicación en este proyecto.

En segundo lugar quiero agradecer a mis compañeros de trabajo por haberme confiado este proyecto y por acompañarme durante todo el proceso de apredizaje.

Por último agradecer a mi familia y amigos su gran ayuda y apoyo incondicional.

RESUMEN

Este proyecto consiste en el desarrollo de un gestor de contenidos para una página web corporativa con un gran tráfico de usuarios.

El desarrollo de un gestor de contenidos propio se fundamenta en dos pilares. El primero consiste en preservar y maximizar la seguridad de la aplicación web mediante un desarrollo privado y el segundo consiste en permitir solamente la edición de determinadas páginas y secciones de la web.

El gestor de contenidos se construye mediante un conjunto de módulos (plugins) independientes que aportan funcionalidad a la aplicación web. Estos módulos se organizan en dos grupos: aquellos que tienen que ver con la administración y gestión del sitio (analítica de datos, copias de seguridad y gestión de archivos) y los que tienen que ver con la gestión del contenido (artículos, sliders, diccionarios, casos de éxito, etc). A su vez, los módulos de gestión de contenido se dividen en dos grupos: los simples y los complejos, en función de si ofrecen la posibilidad de crear páginas nuevas de contenido con URL propia o simplemente una miniatura.

El proyecto se ha programado en Java sobre un framework llamado Play. Se ha diseñado una arquitectura modular y escalable utilizando el patrón MVC (Model, View, Controller) con la integración de servicios de almacenamiento externo y APIs de terceros. El desarrollo se ha llevado a cabo de forma asíncrona para evitar bloqueos en el acceso a la base de datos y garantizar de esta forma el correcto funcionamiento de la web incluso en los momentos de mayor tráfico.

Como resultado se ha conseguido implementar un gestor de contenidos totalmente funcional que cumple los requisitos que se establecieron en el inicio del proyecto y que permite gestionar, mediante nueve módulos desarrollados, gran parte del contenido de la web corporativa.

PALABRAS CLAVE

Gestor de contenidos, programación asíncrona, Play framework, aplicación web modular, MongoDB, representación gráfica de datos, Amazon S3, arquitectura cliente servidor, copias de seguridad.

ABSTRACT

This project consists in the development of a content manager system (CMS) for a corporate website with a large amount of user traffic. The rationale for the development of an own content manager is based on two pillars. The first one is to preserve and maximize the security of the web application through a closed development and the second one is to constrain the pages and sections of the web that can be edited.

The content manager is built using a set of independent modules (plugins) that provide functionality to the web application. These modules are organized in two groups: those related to administration and management of the site (data analytics, backup copies and file management) and those related to content management (articles, sliders, dictionaries, success stories, etc). In turn, content management modules are also divided in two groups: simple and complex, depending on whether they offer the possibility of creating new content pages with their own URL or just a thumbnail.

The project has been developed in Java with a framework called Play. A modular and scalable architecture has been designed using the MVC pattern (Model, View, Controller) with the integration of external storage services and third-party APIs. The development has been carried out asynchronously to avoid blocking database operations, thus guaranteeing performance even in presence of high workloads.

As a result, it has been possible to implement a completely functional content manager that meets the requirements established at the start of the project and allows a content creator to manage, through nine developed modules, much of the content of the corporate website.

KEYWORDS

Content Management System, asynchronous programming, Play framework, modular web application, MongoDB, data graphical representation, Amazon S3, client–server architecture, backups.

1. INTRODUCCIÓN

La finalidad de este proyecto es el desarrollo de un gestor de contenidos para la administración y gestión de una web corporativa enfocada a la publicación de contenidos.

En este capítulo se ponen en contexto los antecedentes que han motivado el desarrollo de este proyecto así como los objetivos que se persiguen. Por último, se detalla el plan de trabajo realizado en cada una de las fases del desarrollo.

1.1. Antecedentes

Este proyecto surge con la necesidad de que un equipo sin conocimientos técnicos pueda gestionar una web corporativa en tiempo real. Hasta el momento, en la empresa, cada actualización de dicha web pasa por las siguientes fases:

- En primer lugar el departamento de *marketing* y comunicación redacta los contenidos que quiere incluir en el sitio web.
- El área de diseño genera un prototipo visual con el diseño y las imágenes que se van a publicar.
- Después, el equipo de desarrollo integra el contenido y el diseño en el proyecto de la web.
- Por último, el departamento de IT lleva a cabo el despliegue de la web en producción.

Este proceso es largo, motivo por el cual solo se actualiza la web una vez por semana. Además, una actualización implica que cuatro departamentos distintos se impliquen en la tarea. En conclusión, se están invirtiendo muchos recursos para una tarea muy sencilla.

El sentido de este flujo de trabajo es por motivos de seguridad. En primer lugar, porque el proyecto que se despliega en producción es un proyecto estático sin lógica en el lado del servidor (*back-end*); solamente contiene recursos visuales de *front-end* como HTML, CSS y JavaScript. En segundo lugar, porque pasando por cuatro fases y cuatro equipos se asegura que el contenido que se publica cumple todos los requisitos requeridos de calidad, marca corporativa y diseño.

Con el fin de reducir el tiempo y el trabajo asociados a este proceso se ha propuesto el desarrollo de un gestor de contenidos para que los equipos de *marketing* y comunicación puedan dar de alta contenidos en tiempo real sin implicar a otros departamentos, logrando así una optimización de tiempo y recursos.

Cuando se analizó el problema se propusieron dos posibles soluciones: la primera implica utilizar un gestor de contenidos existente como Wordpress y la segunda consiste en desarrollar un gestor de contenidos propio. La primera opción se descartó por los dos motivos descritos anteriormente: seguridad y control de calidad. Los gestores de contenidos existentes en el mercado proporcionan mucha funcionalidad y flexibilidad, pero estas características, en exceso, pueden llegar a ser incompatibles con las necesidades de la empresa. Al ser sistemas de código abierto cuya funcionalidad se completa con *plugins* de terceros, la posibilidad de encontrar vulnerabilidades de seguridad es muy alta. Con respecto a la flexibilidad, por requisitos de cumplimiento de calidad y marca corporativa no es posible permitir que toda la web sea editable desde el gestor de contenidos. Por esta razón ha sido necesario desarrollar un gestor propio que permita editar solo determinadas zonas de la web, dependiendo del perfil y categoría del usuario.

1.2. Objetivos

Como se ha dicho anteriormente, el objetivo principal del proyecto es la implementación de una aplicación web orientada a la gestión de contenidos. En esta sección se enumeran los objetivos que se han marcado para el desarrollo de este proyecto.

1.2.1. Gestión de artículos

Implementación de un módulo que permita dar de alta, modificar y eliminar artículos que contengan textos, imágenes y enlaces y que permitan ser categorizados por etiquetas. Todos los módulos de contenido deberán ser susceptibles de especificar una serie de campos obligatorios que serán necesarios para poder publicarse.

1.2.2. Gestión de *sliders*

Implementación de un módulo para gestionar el contenido destacado que aparecerá en la página principal en forma de *banner* con diferentes imágenes (*slides*) que se muestran en sucesión. Como todos los módulos de contenido, incluirá la funcionalidad de publicar y ocultar, así como programar en el tiempo la fecha de la publicación.

1.2.3. Gestión de casos de éxito

Desarrollo de un módulo que permita dar de alta casos de éxito que además incluya la posibilidad de dar de alta nuevas páginas con *URL* propia formadas por otros módulos de contenido (submódulos). Los módulos que conformen dicha página deberán ser independientes y su inclusión será opcional. Como todo módulo con página propia, se deberá implementar la funcionalidad de previsualizar tanto la versión resumida del caso de estudio (miniatura) como su página propia antes de ser publicada.

1.2.4. Gestión de un diccionario tecnológico

Administración de un diccionario tecnológico que permita dar de alta nuevos términos y, en algunos casos, implementar páginas propias asociadas a cada término, con la posibilidad de definir estas páginas mediante un editor de texto enriquecido con un formato visual previamente definido y con la posibilidad de adjuntar imágenes.

1.2.5. Visualización de métricas web

Implementación de un panel de control que muestre las métricas de analítica web más importantes en forma de gráficas aprovechando la *API* de Google Analytics. Además, se realizará una comparativa temporal de los resultados.

1.2.6. Gestión de copias de seguridad

Desarrollo de un módulo que realice copias de seguridad automáticas y permita visualizarlas y restaurarlas en cualquier momento en caso de error humano o fallo técnico.

1.3. Plan de trabajo

En esta sección se enumeran las fases de desarrollo por las que ha ido pasando el proyecto, su estimación temporal y en qué capítulos se desarrollan.

1.3.1. Fase de formación y elección de tecnologías

En esta primera fase se eligieron las tecnologías, tanto del lado del servidor como del lado del cliente sobre las que se ha desarrollado el proyecto. La versión del *framework* web que se ha utilizado ha requerido un tiempo de formación inicial, además de haber ido profundizando a lo largo del proceso de desarrollo al igual que sucede con el resto

de tecnologías. Esta fase se describe en el capítulo 3 y ha requerido un mes de tiempo.

1.3.2. Fase de desarrollo de la arquitectura

Esta fase ha sido la más importante del proyecto y la que más tiempo ha necesitado, aproximadamente cuatro meses. En esta fase se ha implementado una arquitectura asíncrona y modular. Se ha realizado también la integración con un sistema gestor de bases de datos no relacional (MongoDB) y se ha desarrollado un servicio de gestión de ficheros en un servidor de Amazon S3. Esta fase se describe en el capítulo 4.

1.3.3. Fase de desarrollo de módulos sencillos

Una vez definida la arquitectura sobre la que construir los módulos del proyecto, se han implementado los módulos con una funcionalidad más sencilla. El desarrollo de estos módulos ha ido evolucionando hasta llegar a su versión final. Esta fase se describe en la primera sección del capítulo 5 y ha requerido un mes de tiempo.

1.3.4. Fase de desarrollo de módulos complejos

A partir del desarrollo de los módulos simples se ha incrementado la funcionalidad con la posibilidad de definir y publicar páginas propias formadas por submódulos. También se ha añadido la funcionalidad de previsualizar el contenido de estos módulos. Esta fase se describe en el la segunda sección del capítulo 5 y ha requerido un mes de tiempo.

1.3.5. Desarrollo de un sistema de copias de seguridad

Después de definir la arquitectura y los módulos del gestor de contenidos, se ha desarrollado un módulo para realizar y gestionar copias de seguridad de toda la información almacenada en la base de datos. Esta fase se describe en el la segunda sección del capítulo 6 y ha requerido medio mes de tiempo.

1.3.6. Desarrollo de un panel de visualización de datos

Esta fase ha consistido en la exploración de la *API* de Google Analytics para la obtención de datos y métricas relevantes de la web. Además se ha realizado una integración con una librería de código abierto para la representación visual de los datos. Esta fase se describe en el la primera sección del capítulo 6 y ha requerido medio mes de tiempo.

2. INTRODUCTION

The purpose of this project is the development of a content manager system for the administration and management of a corporate website devoted to the publication of content.

In this chapter, the background that has motivated the development of this project as well as the objectives are put in context. Finally, the work plan carried out in each of the development phases is detailed.

2.1. Background

This project arises from the need for a team without technical knowledge to manage a corporate website in real time. So far, in the company, each update goes through the following phases:

- First, the marketing and communication department writes down the contents to be included in the website.
- The design area generates a visual prototype with the design and images to be published.
- Then, the development team integrates the content and design into the web project.
- Finally, the IT department carries out the deployment of the web in production environment.

This is a long process, which is why the web is updated just once a week. In addition, an update implies that four different departments must be involved in the task. In conclusion, many resources are being invested for a very simple task.

The reason for this workflow is security. First of all, because the project that is deployed in production environment is a static project without any logic on the server side (back-end); it only contains visual front-end resources such as HTML, CSS and JavaScript. Secondly, because going through four phases and four teams ensures that the content that is published meets all the quality, corporate brand and design requirements.

In order to reduce the time and work associated to this process, the development of a content manager has been proposed so that the marketing and communication teams can upload content in real time without involving other departments, thus achieving optimization of time and resources.

When the problem was analyzed, two possible solutions were proposed: the first one involves using an existing content manager such as Wordpress and the second one involves developing an own content manager. The first option was discarded for the two reasons described above: security and quality control. The existing content manager systems in the market provide a lot of functionality and flexibility, but these features, in excess, can become incompatible with the needs of the company. Given that these are open source systems whose functionality is completed with third-party plugins, the possibility of finding security vulnerabilities is very high. Regarding flexibility, due to quality compliance and corporate brand requirements, it is not possible to allow the entire web to be editable from the content manager. For this reason it has been necessary to develop an own content manager that allows only certain areas of the web to be edited, depending on the profile and category of the user.

2.2. Objectives

As stated above, the main objective of the project is the implementation of a web application aimed at content management. This section lists the objectives that have been set for the development of this project.

2.2.1. Articles management

Implementation of a module to register, modify and remove articles that contain texts, images and links and that can be categorized by tags. All content modules must be able to specify a series of mandatory fields that will be necessary to be published.

2.2.2. Sliders management

Implementation of a module to manage the highlighted content that will appear on the main page in the form of a banner with different images (slides) shown successively. Like every content module, it will include the publish and hide functionality, as well as the option to schedule the publication date.

2.2.3. Success Stories management

Development of a module to register Success Stories that also includes the possibility of registering new pages with their own URL formed by other content modules (sub-modules). The modules that make up this page must be independent and their inclusion will be optional. Like in any module with its own page, the functionality of previewing both the summary version of the Success Story (thumbnail) and its own page must be implemented before being published.

2.2.4. Management of a technological dictionary

Administration of a technological dictionary that allows new terms to be registered and, in some cases, to implement term-specific pages, with the possibility of defining these pages through a rich text editor with a previously defined visual format and with the possibility of attaching images.

2.2.5. Visualization of web metrics

Implementation of a control panel that shows the most important web metrics in the form of charts taking advantage on the Google Analytics API. In addition, a temporary comparison of the results will be made.

2.2.6. Management of backup copies

Development of a module that automatically makes backup copies and allows an administrator to view and restore them at any time in case of human error or technical failure.

2.3. Work plan

This section lists the phases of development through which the project has gone, its temporal estimation and in which chapters they are explained.

2.3.1. Initial training and choice of technologies

In this first phase the technologies on which the project has been developed were chosen, both on the server side and on the client side. The version of the web framework that has been used has required an initial training time and a subsequent deepening

throughout the whole development process, as it happens with the other technologies. This phase is described in chapter 3 and has required a month of work.

2.3.2. Development of the architecture

This phase has been the most important of the project and the one that has needed more time, approximately four months. In this phase, an asynchronous and modular architecture has been implemented. The integration with a non-relational database management system (MongoDB) has also been carried out and a file management service has been developed on an Amazon S3 server. This phase is described in chapter 4.

2.3.3. Development of simple modules

Once the architecture on which to build the modules of the project has been defined, the modules with a simpler functionality have been implemented. The development of these modules has evolved until they have reached their final version. This phase is described in the first section of chapter 5 and has required a month of work.

2.3.4. Development of complex modules

From the development of the simple modules, the functionality has been extended with the possibility of defining and publishing own pages formed by sub-modules. The functionality of previewing the content of these modules has also been added. This phase is described in the second section of Chapter 5 and has required another month of work.

2.3.5. Development of a backup system

After defining the architecture and modules of the content manager, a module has been developed to perform and manage backup copies of all the information stored in the database. This phase is described in the second section of chapter 6 and has required half a month of work.

2.3.6. Development of a data visualization panel

This phase consisted in the exploration of the Google Analytics API to obtain relevant data and metrics from the web. In addition, an integration with an open source library for visualization of data has been made. This phase is described in the first section of chapter 6 and has required half a month of work.

3. HERRAMIENTAS Y TECNOLOGÍAS

En este capítulo se describen todas las herramientas, tecnologías, lenguajes y librerías que se han utilizado a lo largo del desarrollo del proyecto.

También se especifican todas aquellas que han sido descartadas y los motivos que han motivado esta decisión.

3.1. Herramientas y tecnologías utilizadas

3.1.1. Lenguajes y tecnologías del lado del servidor

Java 8

Java es un lenguaje de programación de alto nivel orientado a objetos que funciona sobre una máquina virtual propia (*JVM*). Una de las principales ventajas por las que se ha utilizado en este proyecto es por la capacidad que tiene de operar en múltiples plataformas.

Concretamente se ha utilizado la versión 8 de Java [4] porque mejora notablemente el soporte para la programación asíncrona respecto a sus versiones anteriores. Esta característica es fundamental para poder conectar el *framework* web Play y el sistema gestor de base de datos MongoDB, que se describirán en las dos siguientes subsecciones. El modelo de programación asíncrono permite evitar bloqueos en la ejecución del servidor cada vez que se realiza una operación de entrada/salida. De este tema se hablará con más detalle en el capítulo 4.

Play Framework

Play [1, 2] es un *framework* de código abierto escrito en Scala que sigue el patrón de diseño MVC (Modelo Vista Controlador). Aunque está escrito en Scala, este lenguaje se compila a *JVM Bytecode*, por lo que se puede utilizar con el lenguaje Java.

Este framework ofrece la gestión completa del flujo de trabajo de una aplicación web, desde que el usuario realiza una petición en la interfaz web, hasta que se procesa, almacena y devuelve una respuesta al navegador web.

Uno de los motivos principales por los que se ha usado este *framework* es que, a diferencia de otros, el código final ejecutado en un servidor de producción es compilado. Esto añade una capa más de seguridad que se marcó como requisito indispensable en la empresa para el desarrollo de este proyecto.

Otro de los motivos importantes para el uso de este *framework* es la capacidad de gestionar de forma óptima la concurrencia. Al ser una aplicación empresarial, el número de usuarios activos en un mismo momento puede llegar a ser muy alto, y la gestión de la asincronía es vital. Por este motivo Play Framework es el candidato ideal gracias al servidor Akka HTTP que integra.

MongoDB

MongoDB [3] es un sistema gestor de base de datos no relacional (*NoSQL*). Este sistema almacena los datos en formato BSON, muy similar a JSON, pero representado como código binario. La organización de los datos se estructura en varios niveles: una base de datos es un conjunto de colecciones que, a su vez, contienen documentos. Cada documento es una entidad de la base de datos, cuya representación no está limitada a seguir una estructura de datos fijada previamente.

Este sistema de almacenamiento proporciona la versatilidad que este proyecto necesita. Es integrable con muchos lenguajes de programación, entre ellos Java. En este proyecto se ha utilizado la versión asíncrona del *driver* de MongoDB para Java.

Amazon Web Services – Amazon S3

Amazon Web Services [6] es un conjunto de herramientas y servicios enfocados al mundo de la computación en la nube. Concretamente, dentro de estos servicios, se encuentra Amazon S3.

Amazon S3 (*Simple Service Storage*) es un sistema de almacenamiento externo que permite gestionar el flujo completo del uso de archivos de una aplicación. En este proyecto se ha utilizado este servicio por varios motivos:

- Permite añadir una capa más de seguridad, al separar físicamente el código que ejecuta la aplicación web y los datos almacenados, evitando de esta forma que pueda subirse un archivo de código malicioso al mismo servidor donde se está ejecutando la aplicación.

- Es fácilmente extensible en cuanto a su capacidad, pues va ampliando el tamaño de almacenamiento bajo demanda.

- Proporciona la posibilidad de crear enlaces firmados. Estos enlaces pueden configurarse con parámetros que especifiquen un tiempo de vida útil de cada enlace, o que sólo sean accesibles a través de un determinado servidor, es decir, nadie podría referenciar estos recursos desde otra aplicación sin conocer las credenciales de acceso.

- Gracias al gran ancho de banda que proporcionan los servidores de Amazon, permite reducir notablemente el tiempo de carga.

Scala

Scala es un lenguaje de programación que combina los beneficios de la programación funcional con la orientación a objetos. Es un lenguaje escalable, con tipado estático y que al estar compilado en *Java Bytecode* tiene la capacidad de ser interoperable con Java.

Play Framework está escrito en Scala. Gran parte de la documentación de Play está únicamente enfocada a este lenguaje, y en partes muy específicas de la aplicación es necesario utilizarlo. Concretamente, el motor de plantillas de este *framework*, llamado Twirl, está basado en Scala. Por tanto, toda la parte relativa a las vistas de la aplicación está escrita en Scala, ya que Twirl es el único motor de plantillas disponible para este *framework*.

JSON/BSON

JSON es una notación específica para almacenar datos estructurados con el formato de objetos de JavaScript.

BSON es el formato que utiliza MongoDB para almacenar los datos. Es muy similar a JSON, pero la representación de los datos se almacena de forma binaria. Aunque, en algunos casos, un documento BSON puede ocupar más espacio que su equivalente en JSON[1] , el primero es mucho más rápido, característica fundamental para la lectura/escritura de datos en tiempo real.

En este proyecto se han utilizado estos formatos para construir un sistema de copias de seguridad que se almacenan en la nube y que pueden ser restauradas en cualquier momento.

1 Explicación de la relación de tamaño entre JSON y BSON: http://bsonspec.org/faq.html

3.1.2. Lenguajes y tecnologías del lado del cliente

HTML 5

HTML (*HyperText Markup Language*) [7] es un lenguaje de marcado que se utiliza para la codificación de páginas web. Define la estructura de un sitio web y los elementos que la componen. Especifica una serie de etiquetas que pueden usarse para declarar una estructura jerárquica estándar que pueda ser interpretada por cualquier navegador web.

CSS3 y SASS

CSS (*Cascade Style Sheets*) es un lenguaje que permite definir el diseño visual de una página web. En resumen, define cómo deben mostrarse los elementos HTML de una página.

SASS [8] es un preprocesador de CSS que permite traducir un código de estilos escrito en un lenguaje extendido (también llamado SASS) en código CSS estándar interpretable por un navegador. El lenguaje SASS ofrece multitud de utilidades que no se encuentran en CSS. Por ejemplo, permite definir variables, funciones, herencia de estilos, anidación de selectores, etc.

Gracias al uso de los preprocesadores de CSS se puede programar código CSS reutilizable y extensible en forma de librería.

JavaScript

JavaScript [9] es un lenguaje de programación no tipado que se ejecuta en el lado del cliente, más concretamente en el navegador web. Permite crear acciones e interactuar con la estructura y elementos de la página web una vez que ésta ha sido cargada y sin necesidad de recibir una orden del servidor.

Aunque, desde hace unos años, es posible ejecutar código JavaScript en el lado del servidor utilizando distintas tecnologías (por ejemplo, Node.js), en este proyecto no se ha utilizado más que en el lado del cliente. La versión de JavaScript que se ha utilizado en el gestor de contenidos es ECMAScript 6.

JQuery

JQuery [10] es una biblioteca de JavaScript que permite unificar las operaciones para que sean compatibles con independencia del navegador, sistema operativo o dispositivo desde el que se ejecute.

Esta biblioteca ofrece una serie de características que facilitan mucho la interacción con los elementos del *DOM* de una página web, y proporciona una forma más sencilla y potente de exprimir las capacidades de JavaScript. Para este proyecto se ha usado la versión 3 de JQuery.

Bootstrap

Bootstrap es una biblioteca multiplataforma para el diseño visual de sitios web y aplicaciones. Su característica más importante es la definición de una rejilla (*Grid*) que permite que una web se adapte a todo tipo de dispositivos (ordenadores, tablets, móviles, etc.), independientemente del tamaño, proporción o sistema operativo que ejecuten.

Además, esta librería proporciona una serie de recursos visuales que facilitan en gran medida la codificación CSS de un sitio web, definiendo todos los elementos básicos que pueden conformar una web: botones, bloques de texto, galerías de imágenes, selectores, ventanas emergentes, paginaciones, etc.

Esta biblioteca, al estar escrita en SASS, se puede usar de dos maneras, en su versión compilada o con el código fuente. En ese proyecto se ha usado directamente integrando el código fuente. De esta forma, Bootstrap permite extender su capacidad programando nuevos componentes, incluyendo sólo las partes de la biblioteca que se vayan a usar, y modificando los parámetros definidos por defecto en el código.
Utilizar Bootstrap de esta manera permite no sólo reducir el peso de la biblioteca, sino ofrecer en cualquier momento el uso de sus funciones, variables y herencia que gracias a SASS hacen de Bootstrap una biblioteca dinámica.

Google Analytics *API*

Google Analytics es una herramienta de analítica web que permite obtener todo tipo de métricas relacionadas con un sitio web.

Proporciona una *API* para JavaScript que permite obtener todos esos datos en tiempo real sin necesidad de ir a consultarlos al portal web de Google Analytics. Gracias al servicio que ofrece esta *API* es posible la creación de un *dashboard* con las métricas más relevantes para los departamentos de marketing y comunicación de la empresa.

Con esta herramienta no sólo es posible obtener los datos, sino filtrarlos, analizarlos y compararlos con otras métricas. Por ejemplo, sería posible comparar el número de visitas de un mes en concreto con el de los últimos 3 años.

3.1.3. Herramientas de ayuda al desarrollo

Sbt

Sbt es una herramienta de compilación y automatización de construcción de proyectos. Gestiona desde la instalación de las dependencias hasta la configuración del proyecto en un servidor de producción.

Esta herramienta permite definir parámetros de configuración como el puerto HTTP en el que se va a ejecutar la aplicación, los datos de acceso a la aplicación y a servicios externos como Amazon S3 y MongoDB.

GitHub

GitHub es una plataforma de almacenamiento para repositorios Git con una interfaz web. Git es un sistema de control de versiones de código. Permite almacenar todas las versiones que se generan del código fuente durante el transcurso del desarrollo del proyecto, ofreciendo la posibilidad de revertir ciertos cambios y conocer en qué estado se encontraba la aplicación en cualquier momento.

Gulp.js

Gulp es un conjunto de herramientas que se utilizan para la compilación en tiempo real de código en el *front-end*. Ofrece un inmenso número de posibilidades para facilitar y automatizar el desarrollo de la parte visual de la aplicación.

Cada una de las herramientas integradas dentro de Gulp recibe el nombre de *plugin*. Los *plugins* se instalan en el proyecto bajo demanda para cubrir determinadas necesidades, como por ejemplo: compilación de código SASS en CSS, minificación de CSS, concatenación de archivos, transpilación[2] de código JavaScript para maximizar la compatibilidad con los navegadores haciendo uso de ECMAScript 6, etc.

2 La transpilación es un proceso que consiste en generar código en un lenguaje de alto nivel a partir de otro código, en este caso para maximizar la compatibilidad con versiones antiguas de navegadores web.

IntelliJ IDEA

IntelliJ IDEA es un entorno de programación para el desarrollo de aplicaciones. Soporta tanto Java como Scala y disfruta de todas las características necesarias durante el proceso de desarrollo: depuración de código, autocompletado, refactorización, etc.

Uno de los motivos más importantes por los que se ha usado esta aplicación es por la integración directa con Sbt. Gracias a esta característica, la configuración del proyecto en el entorno se lleva a cabo de forma muy sencilla.

Atom

Atom es un editor de código abierto multiplataforma que permite la instalación de numerosos plugins que facilitan y aceleran el desarrollo y la escritura de código.

Toda la parte *front-end* de la aplicación se ha desarrollado en este editor utilizando algunos plugins como EMMET para autocompletar fragmentos de HTML, o Beautify para formatear el código JavaScript y CSS.

3.2. Herramientas y tecnologías descartadas

Node.js y Express

Al comienzo del proyecto se valoró la posibilidad de utilizar Node.js como entorno de ejecución para toda la parte *back-end*, junto con Express.js, un *framework* web muy potente diseñado para construir *APIs* y aplicaciones web.

Estas tecnologías se descartaron porque la política de la empresa no permitía utilizarlas por temas de seguridad. Al estar basadas en JavaScript, el código fuente de la aplicación no se compila, sino que se ejecuta por un intérprete. Esto podría dar lugar a la explotación de vulnerabilidades (en particular, la inyección de código arbitrario) si alguien consiguiera acceder al servidor.

Otro de los motivos es que tanto el equipo de IT como el de Seguridad están especializados en configurar y auditar aplicaciones desarrolladas en Java, y el proceso de adaptar un nuevo entorno llevaría más tiempo de análisis por parte de ambos equipos.

React

React es una librería JavaScript de código abierto desarrollada por Facebook con el fin de crear aplicaciones en una sola página. Está basada en la utilización de un *DOM* virtual que se carga a través de JavaScript en tiempo real.

Esta tecnología fue descartada debido a que gran parte de la aplicación tiene que ser indexada por los buscadores, que necesitan leer todo el contenido de la web para posicionarlo. Esto entra en conflicto con la filosofía SPA (*single page application*) que define React, mediante la cual todo el contenido de la página web se construye dinámicamente en el navegador web a partir de la información recibida desde el servidor.

Redux

Redux es una biblioteca de JavaScript que se encarga de gestionar el estado de la aplicación y representarlo mediante otros sistemas, tales como React o Angular. Al descartar previamente React como librería, también se descartó esta biblioteca.

4. ARQUITECTURA DEL SISTEMA

En este capítulo se explica a nivel técnico el funcionamiento interno de la aplicación y su integración con los servicios externos. También se detalla un ejemplo de ejecución para un caso de uso sencillo.

4.1. Qué es Play Framework y cómo funciona

Play Framework es un marco de aplicaciones web que integra y gestiona los componentes y *APIs* necesarias para el correcto desarrollo de una aplicación web. La anatomía de una aplicación Play está basada en el patrón MVC [5] (*Model View Controller*), aunque en este proyecto se ha ampliado la arquitectura para añadir una serie de servicios externos que no se enmarcan en ninguno de estos componentes.

A continuación se describen los elementos más importantes que conforman una aplicación web que utilice este framework.

4.1.1. El paquete "app/"

El paquete *app* contiene en su interior los modelos, las vistas y los controladores de toda la aplicación, a su vez organizados en subdirectorios. En este proyecto se ha optado por utilizar un tipo específico de modelo centrado en los datos, también llamado POJO (*Plain Old Java Object*). Un POJO es una instancia de una clase que no implementa ninguna interfaz ni extiende ninguna clase (salvo Object). Simplemente está formado por un conjunto de atributos básicos y compuestos (siendo, estos últimos, instancias de otros modelos POJO). No obstante, con este *framework* los modelos tienen que implementar una interfaz para la validación, por lo que no se pueden considerar modelos POJO puros. Esta implementación es una restricción del *framework*, ya que la única manera que se ofrece para realizar la validación es implementando la interfaz *Validatable*.

Cada modelo, además de sus atributos, contiene todos los métodos *get()* y *set()* que serán necesarios para realizar correctamente la transformación de un modelo en una entrada de la base de datos, y viceversa.

Por último, la clases modelo tendrán (en caso de ser necesario) la especificación de restricciones adicionales en los valores de sus atributos, de cara a su posterior valida-

ción. Por ejemplo, es posible indicar si un atributo tiene que ser de tipo entero, si tiene que tener una longitud determinada o si es un dato obligatorio.

Todas las operaciones con la base de datos se realizan a través de una clase que implementa el patrón DAO (*Data Access Object*) abstracto[1]. En esta clase se implementan las operaciones más genéricas necesarias para todas las entidades, como por ejemplo: guardar, eliminar, obtener todos los elementos de una misma entidad, etc. Para cada modelo de datos existe una clase que extiende a este DAO abstracto, y que contiene operaciones específicas de ese modelo.

A continuación se incluye un fragmento de código del modelo de datos de una entrada de casos de éxito con validación. La función *validate*() comprueba la validez del objeto. En caso de no ser válido, devuelve una lista con los errores de validación encontrados. Esta lista es vacía en caso de que el objeto haya sido correctamente validado.

```
@Constraints.Validate
public class SuccessStoriesData implements Validatable<...> {

    private ObjectId id;
    private Date publishTime;
    private Boolean published;
    private Boolean detail;
    private List<EnumTags> tags;

    //Thumbnail module
    private ThumbnailSuccessStories t;

    @Override
    public List<ValidationError> validate() {
        final List<ValidationError> errors = new ArrayList<>();
        errors.addAll(t.validate("thumbnail"));
        if (detail != null && detail)
        if (t.getLink().validate("thumbnail.link") != null)
              errors.addAll(t.validate("thumbnail"));
        return errors;
    }

   // getter & setters ...

}
```

1 Abstract DAO: https://www.codeproject.com/Articles/251166/The-Generic-DAO-pattern-in-Java-with-Spring-3-and

Los controladores de la aplicación se encargan de las peticiones de los usuarios, es decir, de comunicar las vistas con el modelo para almacenar y procesar los datos.

El grueso del proyecto se basa en la validación y el envío de formularios de datos asociados a modelos concretos. Gracias a esto se pueden dar de alta multitud de entidades en la base de datos, siempre ligadas a un modelo de datos concreto.

A continuación se incluye un fragmento de código de un controlador asíncrono. En la sección 4.2 se explicará con detalle el concepto de asincronía y su modelo de ejecución.

```
public CompletionStage<Result> delete() {
    // Obtener una estructura DynamicForm con los datos introducidos
    // por el usuario en un formulario web.
    DynamicForm requestData = formFactory.form().bindFromRequest();
    SuccessStoriesDAO sdm = new SuccessStoriesDAO();

    // Eliminar el objeto cuyo identificador sea el campo 'id' del
    // formulario enviado por el navegador.
    return sdm.remove(
        new ObjectId(requestData.get("id")))
        // Después de que se haya realizado el borrado de la BD,
        // redirigir a la URL indicada.
        .supplyAsync(() -> {
              return redirect("/cms/success-stories");
          }, httpExecutionContext.current());

}
```

En una aplicación web basada en la arquitectura MVC, las vistas indican cómo se representa, mediante HTML, el contenido de un modelo. Para ello es habitual el uso de plantillas HTML con marcadores que son posteriormente sustituidos por valores concretos del modelo que quiere representarse. El componente encargado de esta sustitución recibe el nombre de motor de plantillas. Las vistas en este *framework* tienen una característica importante: funcionan sobre un motor de plantillas llamado *Twirl Template Engine*[2] . Este motor utiliza el lenguaje Scala para la definición y el procesamiento de los datos que se quieren representar. A continuación se muestra un ejemplo de una plantilla Twirl:

2 Twirl Template Engine: https://www.playframework.com/documentation/2.7.x/ScalaTemplates

```
@import models.cmsModels.datapedia
@(allTerms: java.util.List[datapedia.DatapediaData], lang: String)

@cms.main("Datapedia") {
      @for(term <- allTerms) {
            <div class="character">
                  @term.getTerm.lang("en").toString().charAt(0)</div>
            <div class="txt-paragraph-large">
                  @term.getTerm.lang("en")</div>
            @if(term.getPublished() != null &&
                  term.getPublished().toString().contains("true")) { ...
            } else { ... }        @helper.form(controllers.cmsControllers.Datapedia.
      update) {
                  @helper.CSRF.formField
                        <input type="hidden" name="id" value="@term.getId()">
                        <button type="submit"></button>
            }
      }
}
```

Twirl permite el uso de unos ayudantes de vista "*helpers*" que facilitan la gestión de los formularios HTML. Aunque existe una amplia variedad de *helpers* predeterminados, se han tenido que implementar algunos ayudantes de vista específicos a la aplicación, con el fin personalizar su representación en código HTML, tanto del elemento correspondiente del formulario como la de los mensajes de error provenientes de la validación. A continuación se muestra uno de los helpers que se ha implementado.

```
@(elements: helper.FieldElements)

    @if(elements.args.get('customType).contains("text")) {

        <div class="input @if(elements.hasErrors){error}">
            <label for="@elements.id">@elements.label</label>
            <div class="w-100">
            @elements.input
            @if(elements.hasErrors){
                  <div class="input-group has-error">
                        <span class="help-block">
                              @elements.errors
                        </span>
                  </div>
            }
            </div>
        </div>
        <div class="info">@elements.infos</div>

    }
```

4.1.2. El paquete "public/"

En este directorio se encuentran todos los recursos estáticos que proveerá el servidor a cada navegador web. En particular, contiene todos los archivos SASS, CSS, JavaScript, imágenes y librerías externas de front-end como JQuery, Bootstrap, etc.

Para procesar los ficheros de este directorio se utiliza la herramienta Gulp.js, como se explica en el capítulo 3. Cuando se ejecuta el proyecto se lanza el *script* de Gulp.js, el cual monitoriza cualquier cambio en los archivos SASS para compilar en tiempo real a código CSS minificado y en los archivos JavaScript para ser transpilado de ECMAScript 6 a código ECMAScript 5, interpretable por todos los navegadores. Esta última tarea se lleva a cabo utilizando el transpilador BABEL.

4.1.3. El paquete "conf/"

Este paquete contiene tres elementos importantes: los archivos de configuración del proyecto, el fichero de rutas y los mensajes de la aplicación en todos los idiomas.

Los archivos de configuración se definen siguiendo el formato HOCON[3]. Contienen variables cuyos valores pueden ser referenciados desde el código fuente de la aplicación. En esta aplicación web, los archivos de configuración habilitan módulos, definen las contraseñas de la aplicación, de la base de datos y de los servicios externos como Amazon S3.

```
include "passwords.conf"
include "it.conf"

# MongoDB
mongo_database=cms

play.modules.enabled += "models.FormattersModule"
play.modules.disabled += "play.data.format.FormattersModule"
play.modules.enabled += "controllers.MyRecurrentTaskModule"
play.i18n.langs = [ "en", "es" ]
# Parameters
backup.time.hours=24
# Secret
play.http.secret.key="..."
# Hosts
play.filters.hosts { ... }
```

3 HOCON format: https://github.com/typesafehub/config/blob/master/HOCON.md

El fichero de rutas, llamado "*routes*", es el encargado de recopilar todas las urls de la aplicación e indicar con qué controlador se conecta cada una de ellas. También define el formato de la url, indicando si es paramétrica. A continuación se muestra un fragmento de este fichero:

```
# Routes
GET             /cms/dashboard                  controllers.Dashboard.show
GET             /datapedia/:page                controllers...show(lang =”en”, page)
POST            /cms/posts/save                 controllers.Post.save
POST            /cms/posts/delete               controllers.Post.delete
POST            /cms/posts/update               controllers.Post.update
```

Por ejemplo, este fichero de rutas indica que cuando el usuario navegue a la *URL* `/cms/dashboard`, el navegador realizará la correspondiente petición, y ésta será atendida por el controlador implementado en el método `show` de la clase `controllers.Dashboard`.

Los ficheros de mensajes son archivos de texto plano con una estructura clave-valor. Contiene todos los identificadores y cadenas de texto que se utilizan en todas las partes de la aplicación.

En este proyecto coexisten dos idiomas, inglés y español, por lo que deben convivir dos archivos de mensajes: "*messages.es*" y "*messages.en*".

Actualmente Play Framework sólo permite el uso de un archivo de mensajes por idioma, y dado el tamaño del proyecto y la perspectiva que tiene de crecer en los próximos años esta limitación supone un problema importante. Utilizar un solo archivo de mensajes implica tener miles de cadenas juntas en un solo lugar que serían muy difíciles de actualizar y mantener.

Para solventar este problema se ha hecho uso de la herramienta Gulp. De esta forma se puede tener infinidad de archivos de mensajes por idioma organizados jerárquicamente en directorios, de tal forma que al detectarse un cambio en alguno de ellos, gracias a una tarea programada en el script de Gulp, se leen todos esos ficheros y se concatenan sus cadenas en un solo archivo final, que es el utilizado por el *framework*.

4.1.4. El archivo "build.sbt"

En este archivo se especifican los módulos y librerías externas, las dependencias del proyecto, las opciones de Java y la versión del *framework* que se quiere utilizar. La herramienta Sbt, mencionada en el capítulo 3, se encarga de procesar este fichero y realizar las tareas de compilación y descarga de dependencias.

4.1.5. Resto de módulos y librerías

El los directorios "lib/" y "project/" se encuentra el código fuente del *framework* y de las dependencias del proyecto. En el directorio "target/" se aloja la aplicación compilada en modo desarrollo y es donde se genera el ejecutable final listo para desplegarse en los servidores de producción.

4.2. Asincronía con Java 8, MongoDB y Play

Este proyecto va a sustituir una web corporativa estática de una empresa multinacional. El número de visitas diarias de la web es muy elevado. Por lo tanto, la gestión de la asincronía y la concurrencia es vital para que el producto final sea viable.

En Java, un hilo, es una instancia del programa que tiene la capacidad de ejecutarse de forma concurrente. El número de hilos de ejecución por defecto que proporciona Play oscila entre 8 y 64. Si se desarrolla una aplicación en la que el acceso a la base de datos es bloqueante (es decir, provoca la detención de la ejecución del hilo mientras éste realiza un acceso a la base de datos) y los 64 hilos disponibles se encuentran ocupados, el usuario número 65 tendría que esperar que algún hilo fuese liberado para poder realizar una petición. En muchos casos la espera sería de apenas unos milisegundos sin importancia, pero si acceden a la vez miles de usuarios el retardo podría ser muy elevado, provocando una pérdida considerable de visitas.

Por este motivo se ha desarrollado el gestor de contenidos siguiendo una política de asincronía. De esta forma se hace un uso mucho más eficiente de esos 64 hilos de ejecución de la aplicación.

Sin una gestión asíncrona, el funcionamiento de una petición es el siguiente: un usuario accede a una página que necesite obtener información de la base de datos, por lo tanto se realiza una petición a MongoDB que tardará 'x' segundos en recuperar los datos solicitados. Una vez recuperados los datos, se envían al navegador para ser visualizados por el usuario. Durante el tiempo 'x' que ha tardado la base de datos en obtener la información, el usuario ha estado bloqueando un hilo de ejecución, haciendo esperar a otros usuarios para iniciar otras peticiones en ese mismo hilo.

Con una gestión asíncrona, cuando un usuario realiza un acceso a la base de datos, esa petición se envía y el usuario queda a la espera de recibir en un **futuro** la respuesta. En ese momento se libera automáticamente ese hilo de ejecución para atender otras peticiones de otros usuarios. Cuando la información de la base de datos ha sido obtenida se crea un aviso y se notifica que la tarea se ha **completado**. En ese momento se le devuelve al usuario el control de un hilo de ejecución que envía los datos obtenidos al navegador web del usuario. De esta forma, durante el tiempo 'x' que ha tardado la

base de datos en obtener la información, el hilo de ejecución no ha estado bloqueado y ha podido atender otras peticiones. Por lo tanto el usuario sólo ha utilizado el hilo de ejecución el tiempo mínimo indispensable, proporcionando al sistema la posibilidad de atender muchas más peticiones simultáneamente.

En el párrafo anterior se han remarcado dos palabras clave en las que se fundamenta la asincronía de todo el proyecto. El concepto de **futuro** especifica que una acción, por ejemplo un método, se encolará para ser ejecutado cuando haya un hilo disponible. El resultado de este método, cuando haya sido **completado**, se devolverá en forma de *callback* a la instancia que lo haya lanzado.

En la nueva versión de Java 8 se ha añadido la clase `CompletableFuture` que implementa la interfaz *Future* y la clase `CompletionStage` que aporta todos los métodos necesarios para sincronizar. Gracias a estas dos herramientas es posible ejecutar funciones asíncronas y esperar a que estas terminen sin bloquear.

Un ejemplo del uso de esta asincronía se puede observar en un controlador que se encargue de representar unos datos obtenidos de MongoDB en una vista HTML. El método del controlador será de tipo `CompletionStage`, y el método que accede a la base de datos será de tipo `CompletableFuture`. Cuando se hayan recuperado los datos se completará el futuro con esos mismos datos. Por otro lado, el método del controlador quedará a la espera de recibir esos datos y, en el momento que los reciba, procederá a renderizarlos en una vista. Mientras tanto, el hilo ha sido liberado para atender otras peticiones.

4.3. Parte privada y parte pública

En este proyecto, como en cualquier gestor de contenidos, conviven dos partes bien diferenciadas: una parte pública accesible para cualquier usuario desde la web y otra parte privada accesible sólo para aquellos usuarios que tengan permisos de administración. En esta última parte se integra el gestor de contenidos propiamente dicho, mientras que en la parte pública se implementa únicamente la visualización de los contenidos almacenados.

El acceso a la parte privada del gestor de contenidos está protegido por un módulo de autenticación desarrollado por la empresa. Éste módulo se integra en el proyecto como un servicio más, pero su desarrollo queda totalmente al margen de este TFG. Este módulo de seguridad proporciona al gestor de contenidos una serie de variables de sesión que son consultadas a lo largo de toda la lógica del proyecto para diferenciar entre tipos de usuarios, roles, acciones permitidas, etc.

4.4. *Action Creator*, el *middleware* de Play

En otros *frameworks* web, como Node.js, es muy común el uso de *middlewares*. Un *middleware* es un fragmento de código que se ejecuta entre la petición de un usuario y el procesamiento del servidor que corresponde a esa petición. Con ello se ofrece la posibilidad de implementar acciones previas (por ejemplo, control de acceso) comunes a todos los controladores.

Play Framework no contempla el uso de *middlewares*, pero proporciona un sistema para interceptar todas las llamadas HTTP que recibe. De esta forma es posible saber si una petición está intentando acceder a la parte pública de la web o la parte privada. En este último caso se comprueba que el usuario esté autenticado y que posea los permisos suficientes para poder concederle el acceso.

Otro uso que se le da a este *Action Creator* es el de analizar en cada petición la *URL* desde la que se solicita. Todas las URLs de la parte pública comienzan con el código de idioma (https://dominio.com/es/directorio). De esta manera es posible obtener el idioma del usuario para cambiar la cookie '*PLAY_LANG*' y servir al usuario los contenidos en el idioma correcto.

4.5. Tareas recurrentes

En este *framework* hay una serie de características que pueden añadirse en el archivo de configuración para activar su funcionalidad. Esta característica surge con la necesidad de ejecutar una tarea de forma periódica para realizar copias de seguridad de la base de datos. La forma en la que Play gestiona esta característica es ejecutando una acción en el momento que se inicia la aplicación. Es configurable a través de tres parámetros:

Delay (tiempo de demora): en el caso concreto de las copias de seguridad se definió como requisito que se hicieran todos los días siempre a la misma hora. Por este motivo se ha hecho uso de la clase `Time` del *framework*. Gracias a esta clase se calcula, mediante una expresión de tipo Cron[4] , el tiempo que falta hasta realizar la próxima copia de seguridad. El resultado de esta operación se introduce en el argumento *delay*.

Intervalo: se define en el archivo de configuración e indica cada cuánto tiempo ha de ejecutarse la acción.

Acción a realizar: en este caso es una llamada a un método que realiza de forma automática una copia de todas las colecciones y documentos de la base de datos MongoDB. Esta característica se explicará con más detalle en el capítulo 7.

4 Expresiones CRON: https://www.playframework.com/documentation/2.7.x/api/java/play/libs/Time.CronExpression.html

4.6. Formatters

Los formateadores proporcionan un flujo de mapeado asociado a un tipo de dato complejo como, por ejemplo, las fechas. En la aplicación, las fechas adquieren un papel importante: es necesario comparar unas con otras, mostrarlas en un formato concreto, etc. Cuando un usuario da de alta un artículo debe introducir la fecha en la que será publicado, pero el formato con el que el usuario introduce una fecha y el formato en que ésta se almacena en la base de datos son muy distintos.

La conversión de ese formato ha de ser transparente para el usuario y para el programador, y es por ello por lo que se ha utilizado la clase `Formatters` de la *API* de Play. En este proyecto se ha desarrollado este *Formatter* para traducir de forma automática, con la ayuda de expresiones regulares[5] , una fecha con un formato intuitivo (12/11/2019 14:56) a una fecha con un formato estándar de Java para almacenarse en la base de datos.

4.7. Sistema de plantillas y localización

En este apartado se describen en detalle tanto el sistema de plantillas que ofrece Play como toda la configuración relacionada con la localización de los idiomas presentes en el proyecto. El sistema de localización que se ha desarrollado en este proyecto es bastante complejo por necesidades de negocio. Las posibilidades que brinda el *framework* no son suficientes para satisfacer todos los requisitos. Por este motivo se ha extendido la funcionalidad del *framework* para cubrir todas las posibilidades.

Twirl, el motor de plantillas de Play, es un lenguaje basado en Scala que proporciona todas las herramientas necesarias para la visualización de los datos. Twirl permite el uso de plantillas heredables para poder construir un sistema modular y mantenible. También ofrece la posibilidad de recibir parámetros como listas, cadenas, variables, objetos, etc. Es posible iterar sobre esta información mediante bucles, utilizar bloques condicionales, declarar fragmentos de código reutilizables, realizar importaciones y escapear HTML.

Este motor de plantillas permite utilizar unos *template helpers* (ayudantes de vista) que facilitan la renderización de campos de formularios en el *front-end*. Además de ser útiles para la visualización se integran con el *back-end* del *framework* para facilitar al programador tareas como la validación y la persistencia de los datos entre peticiones.

Para este proyecto se han implementado *helpers ad-hoc* que se han integrado con componentes externos de *front-end* como editores de texto enriquecido (TinyMCE[6]), selectores de etiquetas o tags, módulos de carga de archivos, etc.

---

5 Expresiones regulares: https://www.regular-expressions.info/posix.html

6 Editor de texto enriquecido TinyMCE: https://www.tiny.cloud/

La localización implementada en el proyecto afecta a todas las partes de la aplicación, desde los modelos a las vistas. A continuación se detalla el impacto que ha tenido el desarrollo en cada una de las partes que se han visto afectadas.

En primer lugar se investigaron las distintas formas de almacenar datos en varios idiomas en una base de datos no relacional. Una alternativa era dividir cada documento en claves de idioma, y dentro de cada una guardar los atributos:

```
artículo: {
      es: {
            title: "Primera forma de localización",
            description: "Descripción del artículo."

      }, en: {
            title: "First localization way",
            description: "Post description."
      }
}
```

Esta opción se descartó por la imposibilidad de almacenar datos de forma genérica que no varíen en cuanto al idioma. Por ejemplo, si se quisiese incluir un nuevo atributo id, cuyo contenido es común a ambos idiomas, sería necesario incluirlo en ambos subdocumentos de manera duplicada. Por otra parte, esta opción hace que el flujo de acceso, filtrado y comparación complique la lógica.

La segunda alternativa, que es la que se implementa en este proyecto, es la de almacenar todos los atributos al mismo nivel, sean del tipo que sean, y dentro de cada uno de ellos incluir las claves de idioma:

```
artículo: {
      title: {
            es: "Primera forma de localización",
            en: "First localization way."
      }, description: {
            es: "Descripción del artículo",
            en: "Post description."
      }
}
```

De esta forma se pueden añadir variables que no dependan del idioma al mismo nivel. Se mantiene en todo momento la consistencia de los datos y el acceso desde el código es muy simple. Esta decisión se ha mantenido a lo largo de todo el proyecto, en los modelos se han extendido los tipos básicos de las variables implementado clases paramétricas que contienen un atributo de tipo <T> por cada idioma definido:

```
public class Language<T> {

    private T es;
    private T en;

    public Language(){}

    public Language(T es, T en) {
        this.es = es;
        this.en = en;
    }
}
```

En los controladores, cada vez que se renderiza una vista HTML, es necesario enviar como parámetro el código de idioma que se ha recuperado, interceptando la petición HTTP correspondiente, y analizando la sintaxis de la *URL* contenida en dicha petición Esto se realiza mediante la clase `ActionCreator`, el mecanismo para implementar *middlewares* en Play.

Para las vistas HTML y las plantillas de la parte pública, además de mostrar el contenido en el idioma correcto, es necesario construir el enlace de cada página en todos los idiomas para que el usuario tenga la posibilidad de cambiar entre ellos. Las páginas estáticas obtienen estos enlaces haciendo uso de "*routing* inverso", mientras que las páginas que se crean de forma dinámica necesitan acceder a la base de datos para consultar sus homólogas en otros idiomas. Por tanto, se ha desarrollado una funcionalidad extra para comprobar si las *URLs* introducidas por los usuarios en las peticiones son válidas o si por el contrario son erróneas y hay que devolver un código 404 y mostrar la página de error.

En el archivo "*routes*" es necesario distinguir cada *URL* por idioma. Por el momento no es posible parametrizar el código de idioma porque el resto de la información de la *URL* también cambia según el idioma.

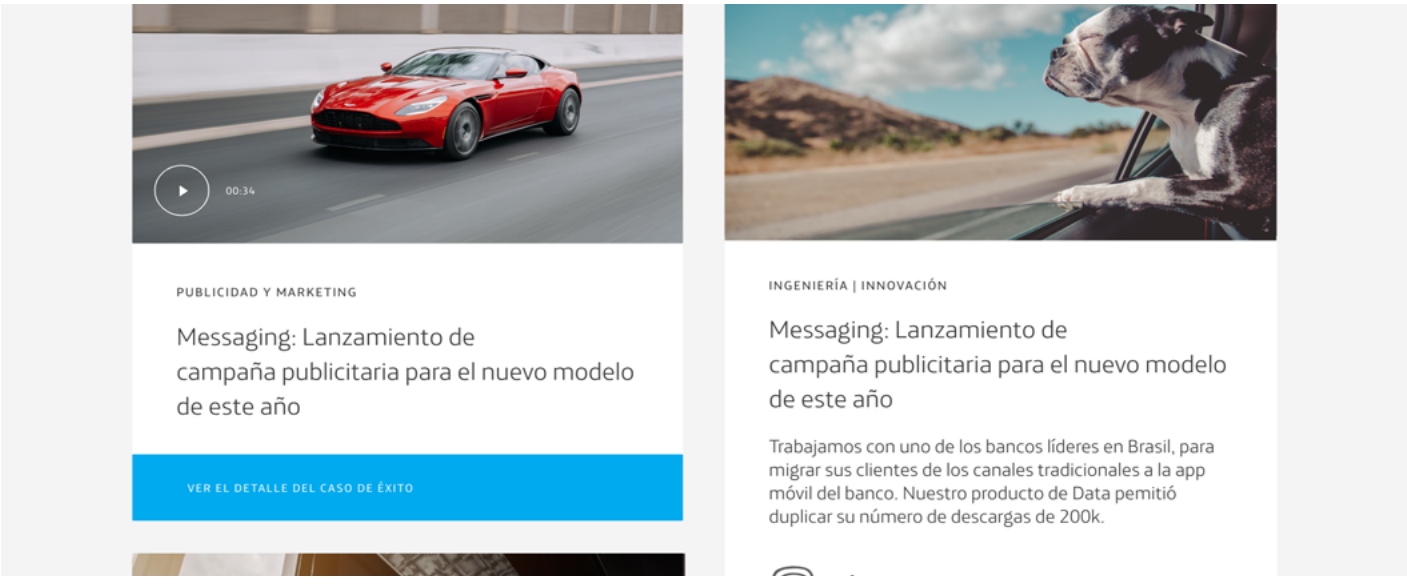
```
GET         /case-studies/:page        ...CaseStudies.show(lang =”en”, page)
GET         /es/casos-de-exito/:page   ...CaseStudies.show(lang =”es”, page)
```

Todo el desarrollo de la arquitectura relacionado con la localización de los idiomas se ha diseñado de forma escalable para que sea extensible a otros idiomas, y para que la incorporación de estos sea lo más sencilla y rápida posible.

4.8. Encapsulación de modelos y vistas

Este gestor de contenidos está desarrollado con la intención de que vaya creciendo y se desarrollen más módulos y componentes para gestionar nuevas áreas de la página web. Por este motivo se ha decidido programar la aplicación de forma que la reutilización de código, la mejora y la creación de nuevos módulos se pueda llevar a cabo de una forma simple.

Cuando se dan de alta contenidos web hay una serie de elementos comunes que se replican por todas las partes de la web. La maquetación del sitio web se ha realizado con el objetivo de poder representar toda la información a partir de una serie de módulos visuales. Estos módulos se replican agrupandose unos con otros para conformar cada página. Cada uno de estos módulos necesitan una serie de información concreta, y en base a esto se han desarrollado tanto componentes de modelos como componentes visuales en plantillas de Twirl. A continuación se indica un ejemplo de un módulo visual y la abstracción de los datos que necesita para completarse. También se especifican los componentes del modelo.



La figura muestra la representación abreviada de varias instancias del módulo casos de éxito. De este ejemplo de módulo se extraen: título, descripción, etiquetas de categorías, enlace, imagen, logotipo y video. Además como cada modelo de datos tendrá un id, una fecha de publicación y un atributo que indica el estado (borrador o publicado).

El primer nivel de encapsulación se lleva a cabo con todos aquellos atributos que son diferentes para cada idioma. En lugar de tener un atributo por cada idioma se ha creado una clase que encapsula todos los idiomas.

```
// Sin encapsulación de idiomas
public class ThumbnailSuccessStories {
    private String title_es;
    private String title_en;
    private String description_es;
    private String description_en;
}
```

```
// Con encapsulación paramétrica
public class ThumbnailSuccessStories {
    private Language<String> title;
    private Language<String> description;

}
```

El segundo nivel de encapsulación se produce con elementos básicos que contienen otros atributos dentro. Por ejemplo, un enlace siempre contendrá un texto, la *URL* del elemento enlazado, un título para el SEO y un atributo que indique si tiene que abrirse en la misma pestaña o en una nueva. Una imagen siempre tendrá el enlace clave a Amazon S3, un título y una descripción textual de la imagen. Estos dos últimos campos tienen como finalidad el posicionamiento en buscadores (SEO).

```
// Sin encapsulación de componentes básicos

public ThumbnailSuccessStories() {}

    private Language<String> text;
    private Language<String> link;
    private Language<String> seoTitle;
    private LanguageB tab;


}
```

```
// Con encapsulación de componentes básicos

public ThumbnailSuccessStories() {}

    private Link link;

}

public class Link {

    private Language<String> text;
    private Language<String> link;
    private Language<String> seoTitle;
    private LanguageB tab;
}
```

Por último, el tercer nivel de encapsulación se produce a nivel de módulos. Al ser un componente que se replicará en muchas páginas distintas tenerlo encapsulado en una clase facilitará la tarea de incluirlo donde se requiera.

```
// Encapsulación de componentes complejos

public class SuccessStoriesData {

    // Review module
    private ReviewData review;

    // Otros módulos
    // ...

}
```

```
// Componente complejo

public class ReviewData {

    private Boolean show;
    private Language<String> client;
    private Language<String> job;
    private Image logo;
    private Link personalWeb;

}
```

De la misma manera esta encapsulación se lleva a cabo en las vistas. En estos tres niveles se crean plantillas que se van heredando para construir el componente final.

4.9. Amazon S3 y carga asíncrona de ficheros

La gestión de archivos adjuntos a los elementos de todo el gestor de contenidos (por ejemplo, imágenes) se ha externalizado utilizando *Amazon Web Services*, en concreto el servicio Amazon S3, como se detalla en el capítulo 3.

La decisión de utilizar este servicio se ha tomado por requisitos de seguridad, ya que de esta forma no es posible subir un fichero al mismo servidor donde se está ejecutando la aplicación ni al servidor donde se alojan las bases de datos. Por lo tanto, se añade una capa más de seguridad evitando que se pueda llegar a ejecutar un archivo malicioso en el servidor.

Con este propósito se ha desarrollado, dentro de la misma aplicación, una *API REST* que proporciona un servicio de carga de ficheros de forma asíncrona a través de peticiones externas. En todos los módulos del gestor de contenidos donde se permite subir un fichero se hace una llamada mediante *AJAX* a esta *API REST*. Este servicio analiza si el tipo de archivo es el correcto y si no excede el tamaño máximo permitido. Posteriormente realiza la carga en Amazon S3 y en caso de éxito devuelve un enlace con la ruta que apunta al fichero. Este enlace es el que se almacena en la base de datos para ser referenciado cuando sea necesario.

De esta forma, toda la carga de archivos pasa por el mismo sitio: el *endpoint* de la *API*. Este sistema se ha desarrollado de esta forma porque por motivos de seguridad se requería centralizar la subida, y porque en un futuro cercano se desea integrar este módulo con otra *API REST* de la empresa para limpiar los metadatos de cada fichero antes de subirlo.

4.10. El *front-end*

El *front-end* de la aplicación es una parte muy importante del gestor de contenidos. Este proyecto se ha desarrollado dentro de un departamento de marca corporativa, es decir, una división que define el estilo y la imagen de la empresa en medios digitales.

Para llevar a cabo de manera correcta y eficiente esta tarea y con la idea de desarrollar un portal basado en el concepto de *Web Components* se ha desarrollado una librería *front-end* que define todos los estilos necesarios para el correcto desarrollo de la web. Esta librería integra la versión en código fuente de Bootstrap, modificando parámetros del *grid*, tamaños, márgenes, etc.

Además se ha implementado, mediante el preprocesador SASS, una serie de herramientas para facilitar el uso de esta librería. En particular, se ha incorporado una sección de colores que contiene los códigos hexadecimales en forma de variables para que puedan ser usados durante el desarrollo, una definición estándar de estilos para las diferentes jerarquías de texto especificadas, una serie de elementos comunes prediseñados como botones, encabezados, etc. También incluye una definición de funciones que pueden ser extendidas desde cualquier archivo para optimizar la reutilización de código.

4.11. Flujo completo de un caso de uso sencillo

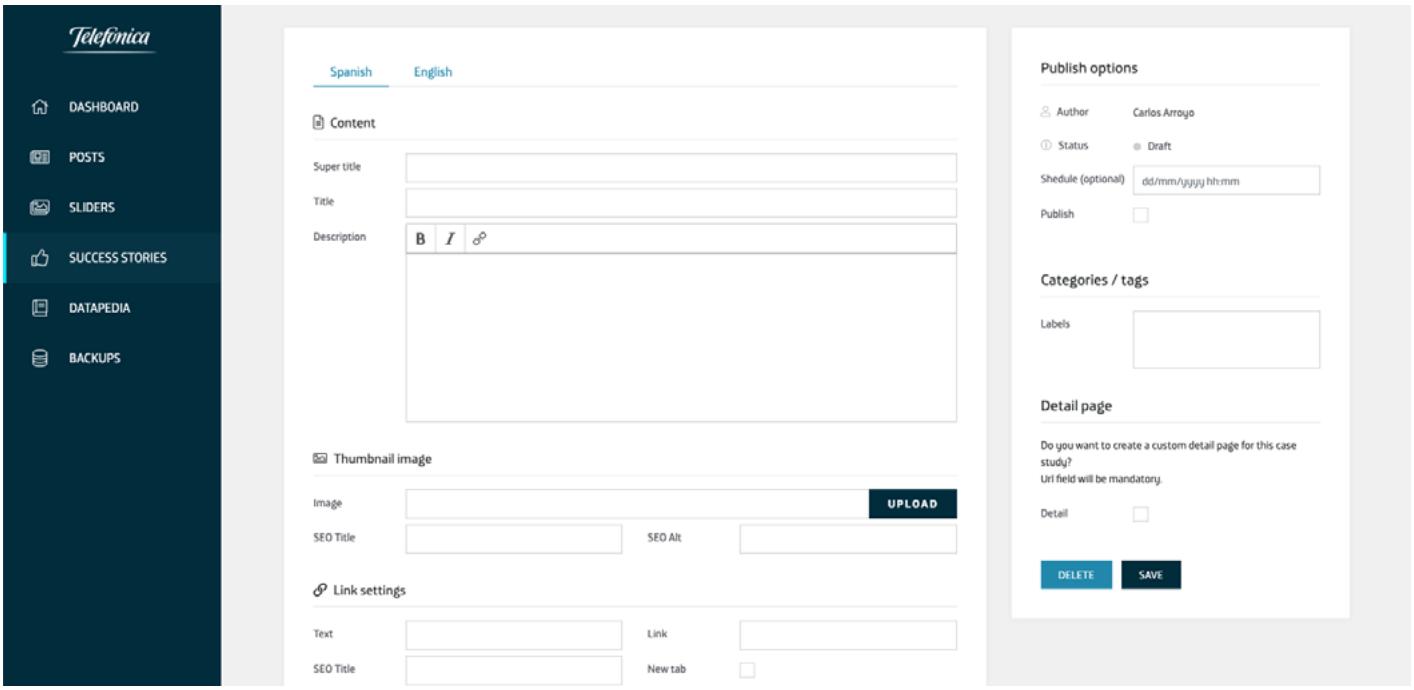
Una vez detallado el funcionamiento de cada uno de los componentes de Play, y con el fin de ofrecer una visión general de dicho *framework*, esta sección describe el flujo completo de ejecución asociado a un caso de uso sencillo del gestor de contenido. En concreto se detalla el proceso de alta de un artículo.

En primer lugar, el usuario introduce en el navegador la *URL* que lleva al formulario de alta de artículos. El navegador envía al servidor una petición de tipo *GET* con esa *URL* y la aplicación web coteja con el archivo de *routes* si esa dirección está asociada a alguna acción, es decir, si hay un controlador encargado de procesar esa petición.

```
GET         /cms/posts/save         controllers.Post.create
```

En este caso el método *create*() del controlador de *Post* envía al navegador la página con el formulario de alta de un artículo.

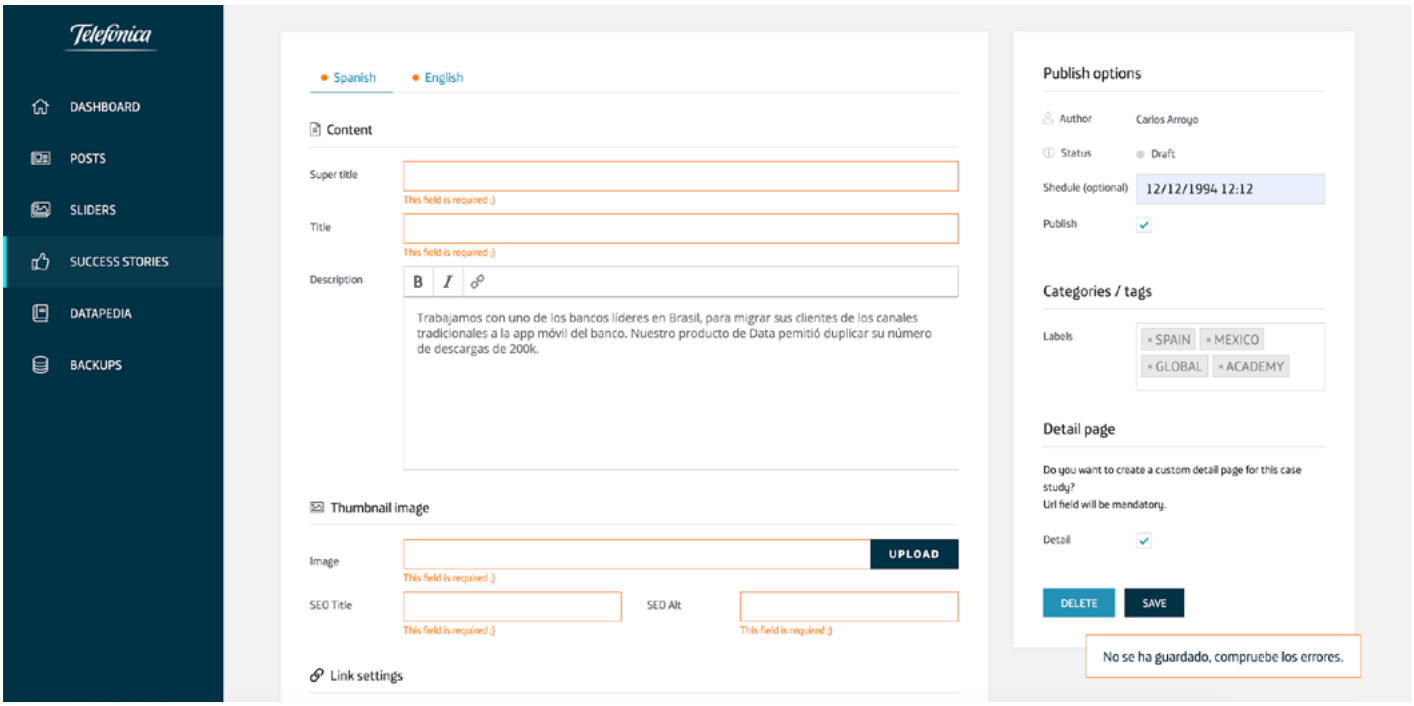
```
public Result create() {

    Form<PostData> postForm = formFactory.form(PostData.class);
    postForm = postForm.fill(new PostData());

    return ok(views.html.cms.posts.addPost.render(postForm));
}
```



El usuario rellena los datos del formulario que sean necesarios para el alta del artículo y hace click en el botón de guardar. Es este momento el navegador envía una petición de tipo *POST* al servidor. Nuevamente se comprueba si en el archivo de *routes* hay algún controlador que se haga cargo de atender la petición, en este caso el método *save*() del controlador *Post*.

```
public CompletionStage<Result> save() {
    PostDAO pm = new PostDAO();
    Form<PostData> postForm = formFactory.form(PostData.class);
    final Form<PostData> boundForm = postForm.bindFromRequest();

    if (boundForm.hasErrors()) {
      final CompletableFuture<Form<PostData>> future = new CompletableFuture<>();
            future.complete(boundForm);
          flash("error", "No se ha guardado, compruebe los errores.");
      return future.thenApplyAsync(result ->
            ok(views.html.cms.posts.addPost.render(result)),
            httpExecutionContext.current());
    } else {
      PostData data = boundForm.get();
          flash("success", "Guardado correctamente");
          return pm.save(data, data.getId()).thenApplyAsync(result -> {
              Form<PostData> newPostForm = formFactory.form(PostData.class);
              newPostForm = newPostForm.fill(result);
              return ok(views.html.cms.posts.addPost.render(newPostForm));
          }, httpExecutionContext.current());
    }
}
```

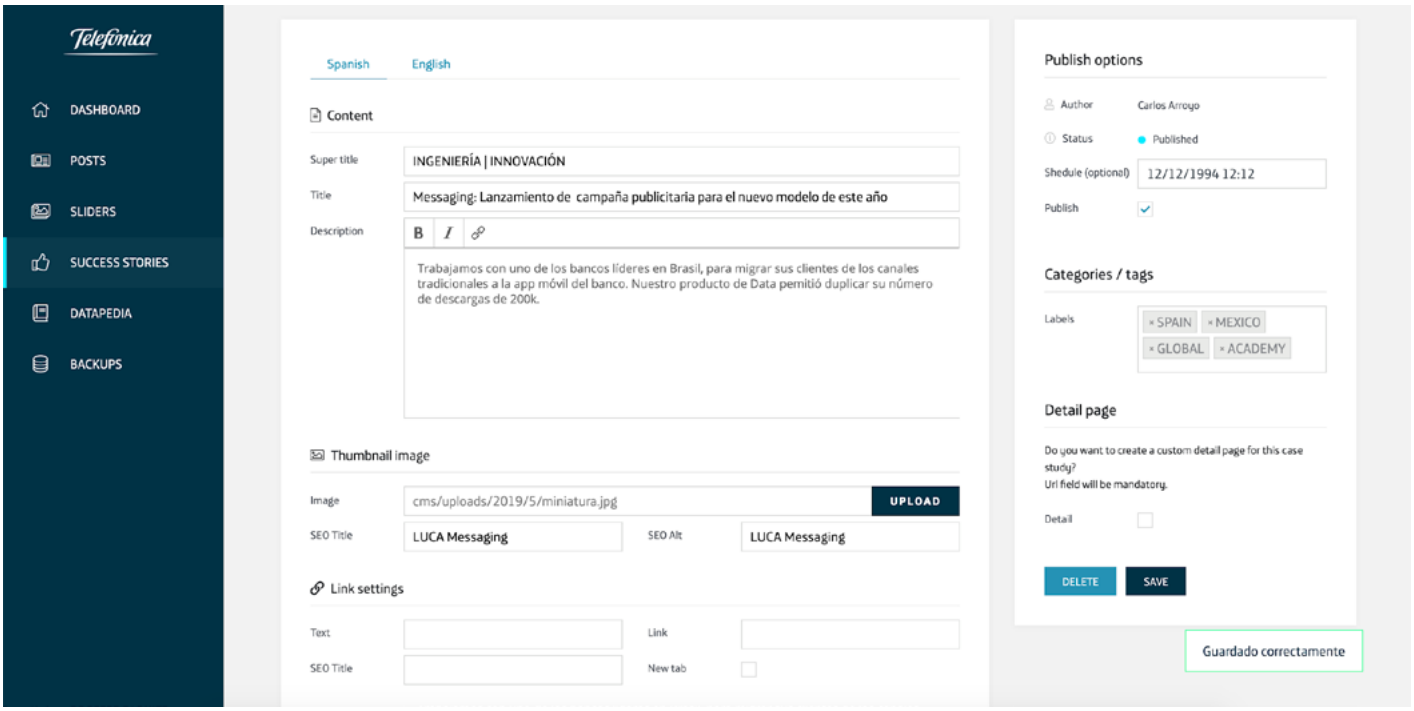
En este método asíncrono se hace la transformación de la información recibida en el formulario con el modelo de datos asociado, después se comprueba si hay algún error que incumpla las restricciones impuestas por el modelo. En caso de que haya algún error se vuelve a representar el formulario con todos los datos que el usuario haya introducido, indicando los que son erróneos.



Si todos los datos son correctos se llama al método *save*() del *DAO*, que de manera asíncrona, como se explica en la sección 4.2, guarda en la base de datos un documento con la información proporcionada.

```
public CompletableFuture<T> save(T t, ObjectId id) {

    MongoCollection<T> collection = getCollection();
    final CompletableFuture<T> future = new CompletableFuture<>();

    // Check if exists
    SingleResultCallback<T> callbackCheck = (result, e) -> {
        if (result == null) { // If not exists create new one
            collection.insertOne(t, (insertResult, insertError) -> {
                if (insertError == null)
                    future.complete(t);
                else
                    uture.complete(null);
            });
        } else { // If exists update }

    collection.find(eq("_id", id)).first(callbackCheck);
    return future;
}
```

Cuando la tarea asíncrona termina, se devuelve al usuario un mensaje que indica que la acción se ha llevado a cabo de forma correcta.



5. MÓDULOS DEL GESTOR DE CONTENIDOS

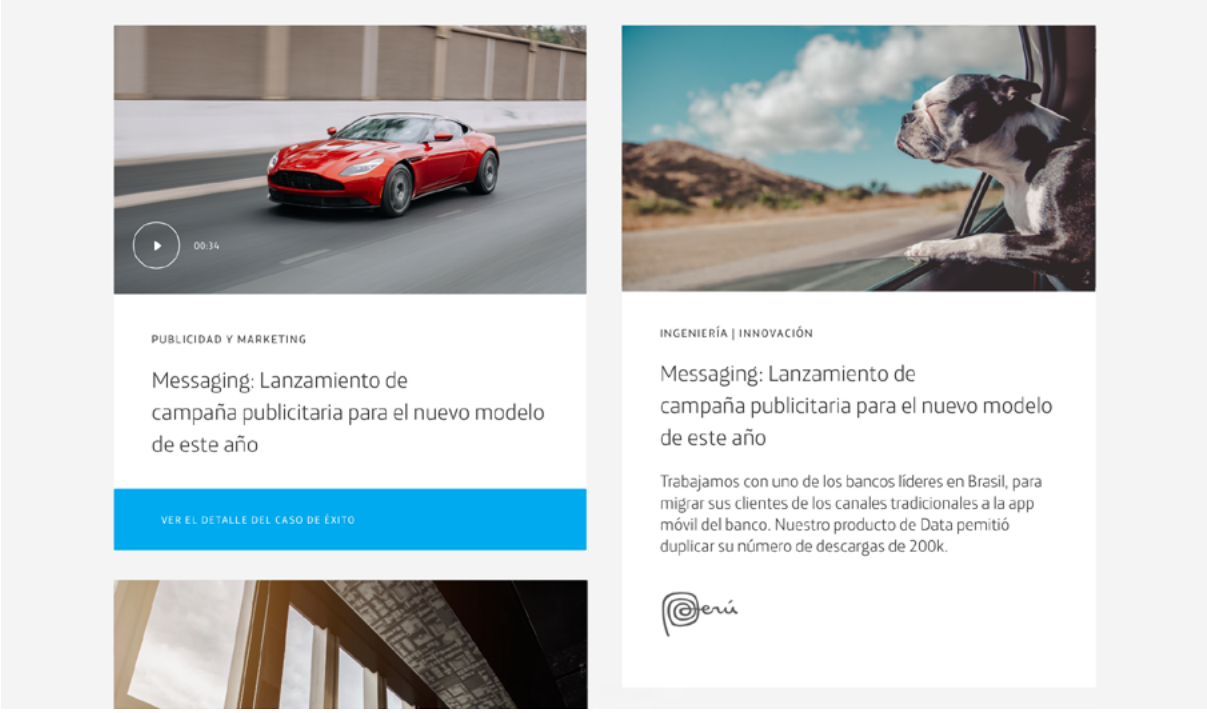
En este capítulo se explica qué es un módulo del gestor de contenidos, los tipos de módulos existentes y cómo crear nuevos para extender la funcionalidad del proyecto.

Un módulo es una sección del gestor de contenidos que permite administrar de forma completa un componente de la página web. Proporciona la funcionalidad de crear, actualizar, eliminar y acceder a cada una de las entidades individuales que forman parte del componente. Por ejemplo, el módulo *Posts* permite administrar todas las noticias de la página web. En este gestor de contenidos, un módulo podría equipararse a un *plugin* de otros gestores como Wordpress[1].

Se han organizado los módulos en dos grupos: simples y complejos. Los primeros constituyen una entidad única que contiene una información fija que será mostrada en un solo lugar de la aplicación web. Los módulos complejos son una extensión de los simples que permiten asociar a un módulo simple mucha más información, que será mostrada en otra página distinta del portal web. Un ejemplo de módulo simple es el módulo *Posts*, constituido por una serie de datos que se mostrarán en la página de "Últimas noticias" a modo de miniatura.

1 Wordpress: https://es.wordpress.com/

Un ejemplo de módulo complejo es el módulo *Success Stories* (casos de éxito). Un caso de éxito es una colaboración con otra empresa en la que se ha implantado un producto nuevo y ha tenido una repercusión positiva en los resultados de la empresa. Este módulo tiene las mismas características de un módulo simple, pero además añade la posibilidad de crear una página de detalle adicional, con una *URL* propia y con más información.



A continuación se especifica el funcionamiento y las posibilidades de cada tipo de módulo y se explican los módulos concretos que se han desarrollado.

5.1. Módulos simples

Dentro de la categoría de módulos simples tenemos los módulos *Posts* y *Sliders*. El módulo *Posts* permite crear miniaturas para ser representadas de un modo similar al listado de publicaciones de un *blog*. Permite reunir en una página diversos enlaces a noticias, eventos, etc.. El módulo *Sliders* permite gestionar el contenido de un *banner* en el que se muestran sucesivamente distintas imágenes (*slides*), cada una de ellas con un texto propio.

La lógica de ambos módulos en cuanto a su funcionamiento es similar. Por este motivo se describe únicamente en detalle el módulo *Posts*. Las características básicas de un módulo simple son las operaciones *CRUD (Create, Read, Update & Delete)*.

Los datos necesarios para la creación de una publicación están definidos en su modelo de datos. Son los siguientes:

```
@Validate
public class PostData implements Validatable<ValidationE-
rror> {

    private ObjectId id;
    private Language<String> title;
    private Language<String> content;
    private Image image;
    private Link url;
    private Language<String> seoTitle;
    private Language<String> seoDescription;
    private Language<String> seoKeywords;
    private Date publishTime;
    private List<EnumTags> tags;
    private String startDate;
    private String endDate;
    private EnumTags e;
    private String location;
    private Boolean published;
}
```

Todos los atributos de tipo *Language* contienen, a su vez, tantos atributos del tipo paramétrico (en este caso, cadenas de texto) como idiomas soportados por el gestor. Además, los tipos Image y Link son objetos que contienen más atributos en su interior, como se detalla en el capítulo 4.

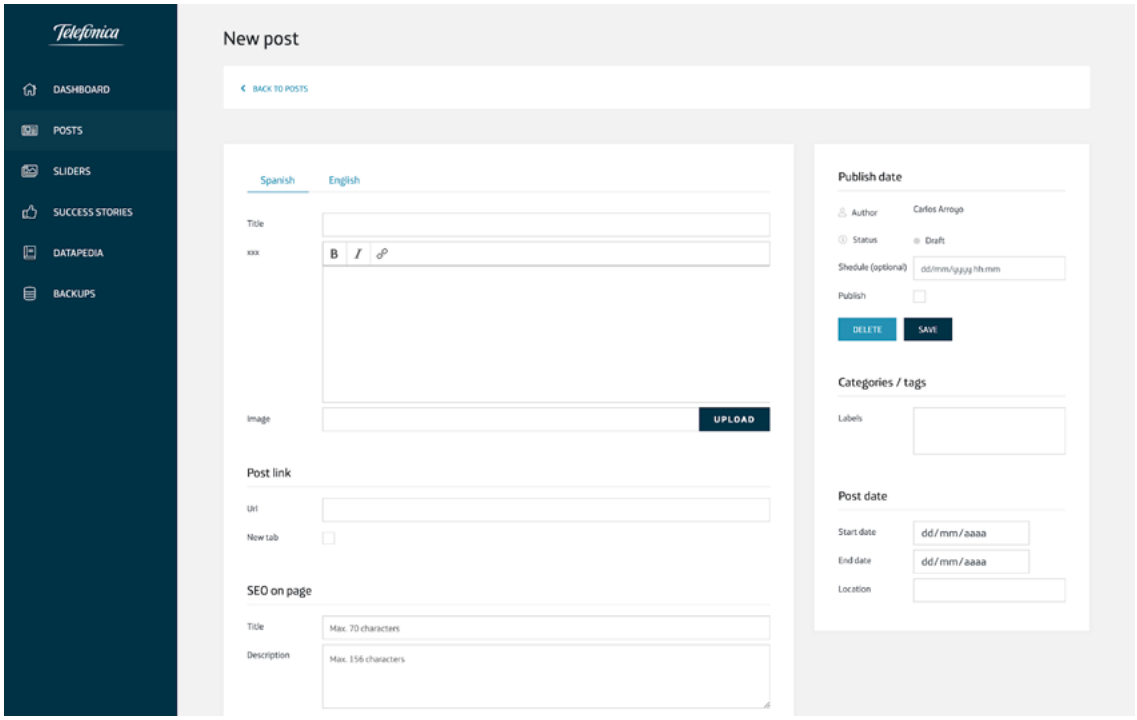
Todos los modelos, además de la definición de sus atributos, contienen los métodos *get*() y *set*() de cada atributo para poder establecer la asociación con los atributos correspondientes de la base de datos. Además implementan un método *validate*() que se encarga de la validación de esos datos en el lado del servidor. Cada modelo hace uso de un *DAO* propio (que extiende, a su vez, al *DAO* abstracto) para realizar las operaciones con la base de datos. En el caso de necesitar operaciones específicas de acceso a la base de datos, los métodos necesarios se implementan en el *DAO* de *Posts*.

```
public class PostDAO extends AbstractDAO<PostData> {

    protected final static String COLLECTION_NAME = "posts";
    @Override
    protected String getCollectionName() {
        return COLLECTION_NAME;
    }

    @Override
    protected MongoCollection<PostData> getCollection() {
        MongoCollection<PostData> collection =
        MongoDB.getDBInstance().getCollection(getCollectionName(),
        PostData.class);
        return  collection;
    }
}
```

Una vez definidos los atributos necesarios en el modelo se detallan las operaciones que se pueden realizar con estas entidades, en concreto, las publicaciones.

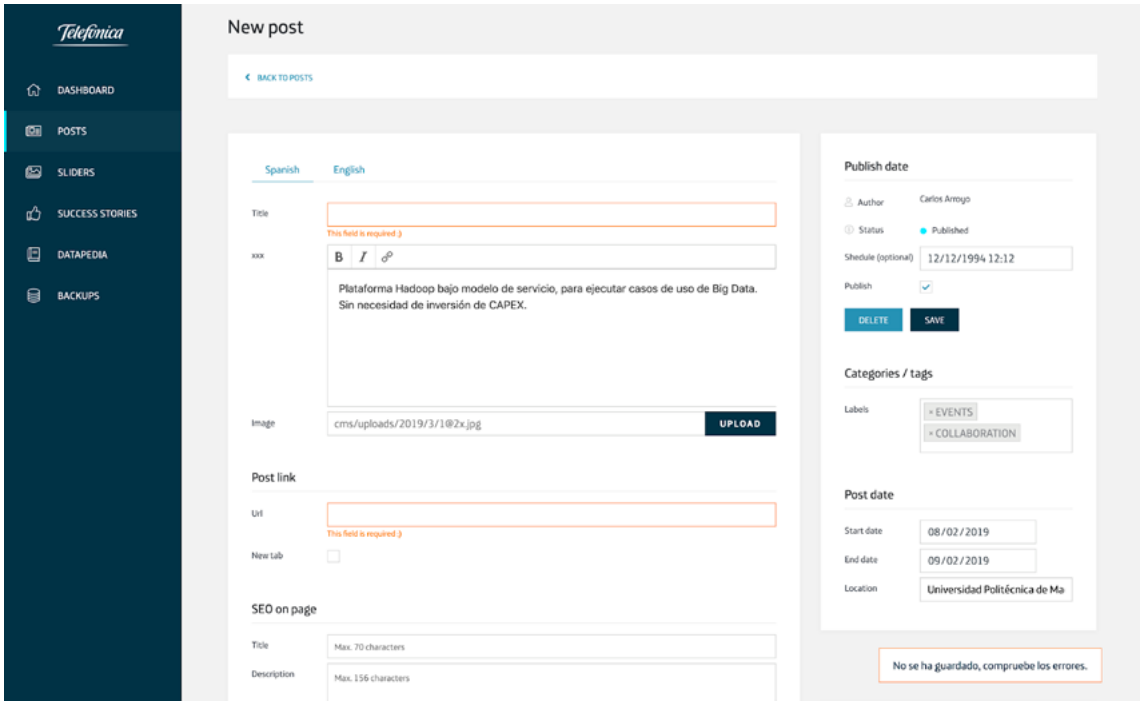
5.1.1. Dar de alta una nueva publicación

Para crear una publicación será necesario mostrar la pantalla con el formulario de creación, la cual se obtiene mediante una petición de tipo *GET* a la *URL /cms/posts/save*. En este momento el usuario tiene a su disposición un formulario para rellenar los campos de información que conforman una entidad de este tipo.



Cuando el usuario introduzca la información de cada campo en ambos idiomas, ejecutará la acción de guardar mediante una petición de tipo *POST* a la misma URL */cms/posts/save*. En el controlador de este módulo se hace una transformación de los datos introducidos en el formulario para obtener un modelo PostData. Si se detecta algún error en los datos, se devuelve al usuario la información que había rellenado en la última petición indicando la existencia de errores y mostrando el error específico en cada campo. En el caso de que todos los datos sean correctos, se llamará desde el controlador al método *save*() del *DAO* de *Post* para escribir en la base de datos la información introducida. Por último, se vuelve a mostrar la misma pantalla del formulario de alta con los datos introducidos por el usuario, por si este quiere seguir editando los datos.

```
public CompletionStage<Result> save() {

    PostDAO pm = new PostDAO();
    Form<PostData> postForm = formFactory.form(PostData.class);
    // Se obtiene la información del formulario
    final Form<PostData> boundForm = postForm.bindFromRequest();

    if (boundForm.hasErrors()) {
// En caso de error se muestra el formulario con el mismo modelo
     final CompletableFuture<Form<PostData>> future =
             new CompletableFuture<>();
             future.complete(boundForm);
        flash("error", "No se ha guardado, compruebe los errores.");
             return future.thenApplyAsync(result -> ok(views.html.cms.
             posts.addPost.render(result)), httpExecutionContext.cu-
             rrent());
        } else {
             // Si la información es correcta se obtiene el modelo
      // correspondiente a la información enviada en el formulario
             PostData data = boundForm.get();
             flash("success", "Guardado correctamente");
             // Se guarda la nueva publicación en la base de datos
        return pm.save(data, data.getId()).thenApplyAsync(result -> {

        Form<PostData> newPostForm = formFactory.form(PostData.class);
                // Se genera un nuevo formulario con la información
                // recién insertada
                newPostForm = newPostForm.fill(result);
             // Se envía el formulario con los datos al usuario
        return ok(views.html.cms.posts.addPost.render(newPostForm));
             }, httpExecutionContext.current());
        }
    }
```



5.1.2. Editar una publicación existente

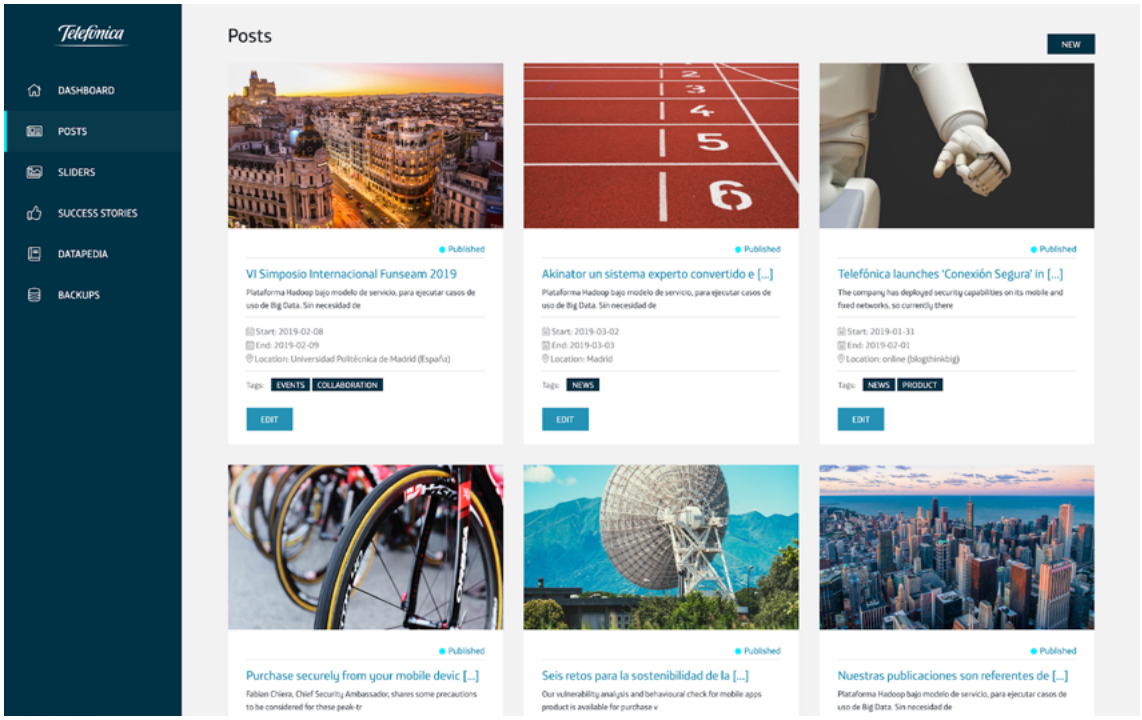
Editar una publicación es muy similar a crear una nueva. El primer paso es seleccionar la publicación que se quiere editar. Después el controlador obtiene la información de la base de datos correspondiente a la publicación seleccionada y la representa en los formularios. A partir de ahí el flujo funciona exactamente igual que en la sección anterior, es decir, llamando al método *save*() del *DAO*.

```
public CompletionStage<Result> update() {

    PostDAO pm = new PostDAO();
    DynamicForm requestData = formFactory.form().bindFromRequest();

    return pm.getOne(new ObjectId(requestData.get("id")))
      .thenApplyAsync(news -> {
            Form<PostData> postForm = formFactory.form(PostData.
class);
            postForm = postForm.fill(news);
            return ok(views.html.cms.posts.addPost.render(postForm));
    }, httpExecutionContext.current());
}
```

5.1.3. Listar todas las publicaciones

Esta acción se realiza haciendo una petición de tipo *GET* a */cms/posts*. A nivel de código, es uno de los casos de uso más simples: se obtienen todos los *posts* que existen y se visualizan a modo de tablón (*dashboard*) para poder ser editados.

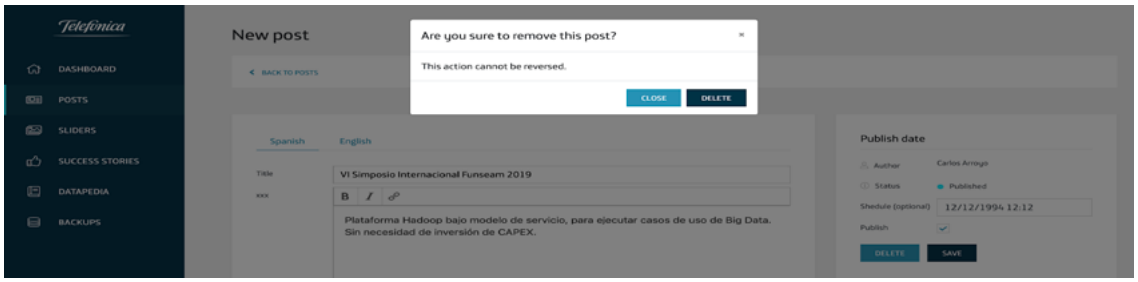
```
public CompletionStage<Result> getAll() {

      PostDAO pm = new PostDAO();
return pm.getAll().thenApplyAsync(list ->
            ok(views.html.cms.posts.news.render(list,
"es")),
            httpExecutionContext.current());
}
```



5.1.4. Eliminar una publicación

La eliminación de una publicación se realiza dentro de la pantalla de edición, independientemente de si la publicación ya existía y se estaba editando, o de si la publicación se acaba de crear. Para eliminarla es necesario hacer *click* en un botón que abre una ventana modal que requiere la confirmación de la acción, para evitar eliminar una publicación por error. En el momento que se confirma la acción, la publicación se borra de forma permanente de la base de datos.

En la acción de eliminar se presentan dos variantes. La primera y más simple consiste en borrar solo el contenido de la base de datos. La segunda variante, utilizada en este módulo, además de borrar la publicación de la base de datos, borra previamente las imágenes asociadas en el servidor de Amazon S3.




```
public CompletionStage<Result> delete() {

 DynamicForm requestData = formFactory.form().bindFromRequest();
     PostDAO pm = new PostDAO();
     return pm.getOne(new ObjectId(requestData.get("id")))
      .thenApplyAsync((result) -> {

         if (result != null) {
            if (result.getImage().getEs() != null &&
                !result.getImage().getEs().isEmpty()) {
                // delete old image if exists
              AmazonS3Utils.delete(result.getImage().getEs());
            }
            if (result.getImage().getEn() != null &&
                !result.getImage().getEn().isEmpty())
                // delete old image if exists
              AmazonS3Utils.delete(result.getImage().getEn());

              pm.remove(new ObjectId(requestData.get("id")))
                  .supplyAsync(() -> {
                        return true;
               });
         }
         return redirect("/cms/posts");
     }, httpExecutionContext.current());
}
```

Los *posts*, además de las operaciones *CRUD*, tienen algunas características añadidas. En particular, pueden ser clasificados mediante etiquetas previamente definidas. Esta característica permite que con el mismo modelo de datos este módulo sea polivalente, es decir, puede ser utilizado tanto para noticias como para eventos, miniaturas de productos, investigaciones, etc. Esto se debe a que los atributos que componen una publicación son claramente extrapolables a un gran número de componentes.

Otra característica importante de todos los módulos es la posibilidad de publicarlos o mantenerlos en estado de borrador mientras son editados, revisados y aprobados. Como añadido, se ha implementado la funcionalidad de programar la publicación en el futuro. Por ejemplo, si la empresa quiere notificar un evento importante que no se hará público hasta el domingo, el usuario que lo dé de alta puede introducir la fecha del domingo como día de publicación, para que automáticamente el domingo aparezca en la parte pública de la web.

5.2. Módulos complejos

Los módulos complejos heredan toda la lógica y el funcionamiento de los módulos simples, pero además añaden un elemento diferenciador: la posibilidad de generar una página de detalle vinculada a una entidad. Esta posibilidad es opcional: en un módulo complejo pueden coexistir elementos sin página de detalle y elementos con ella.

En este apartado se detalla el funcionamiento del módulo *Success Stories*, aunque también se ha desarrollado el módulo *Datapedia*, un diccionario tecnológico que contiene términos relacionados con *Big Data e Inteligencia Artificial*.

El modelo de datos del módulo *Success Stories* difiere en dos aspectos de los módulos simples. El primero es la incorporación de un atributo detail de tipo *booleano* que indica la presencia de una página de detalle. El segundo es la definición de los submódulos contenidos en esta página de detalle (en caso de existir). A partir de este momento, además de dar de alta una entidad, se necesita construir una página de detalle. Por tanto, se necesita introducir información adicional. Esta información, que podrá ser variable, está formada por una serie de secciones cuya información se cumplimentará con los submódulos.

Cada submódulo constituye los datos de una sección, y se encuentran encapsulados tanto sus modelos como sus vistas. Por tanto, en el modelo de *Success Stories* se definen todas instancias a los submódulos que contiene.

```
@Constraints.Validate
public class SuccessStoriesData implements
      Constraints.Validatable<List<ValidationError>> {

      // Common attributes ...
      private Boolean detail;

      // Submodules
      private ThumbnailSuccessStories thumbnail;
      private BannerSuccessStories banner;
      private SummaryData summary;
      private AboutData about;
      private AboutColsData aboutCols;
      private ChallengeData challenge;
      private ImageModule imageModule;
      private ImageModule imageFull;
      private VideoModule video;
      private WhyModule why;
      private ReviewData review;
      private IconsModule productIcons;
      private IconsModule featureIcons;
      private CheckListModule checkList;
}
```

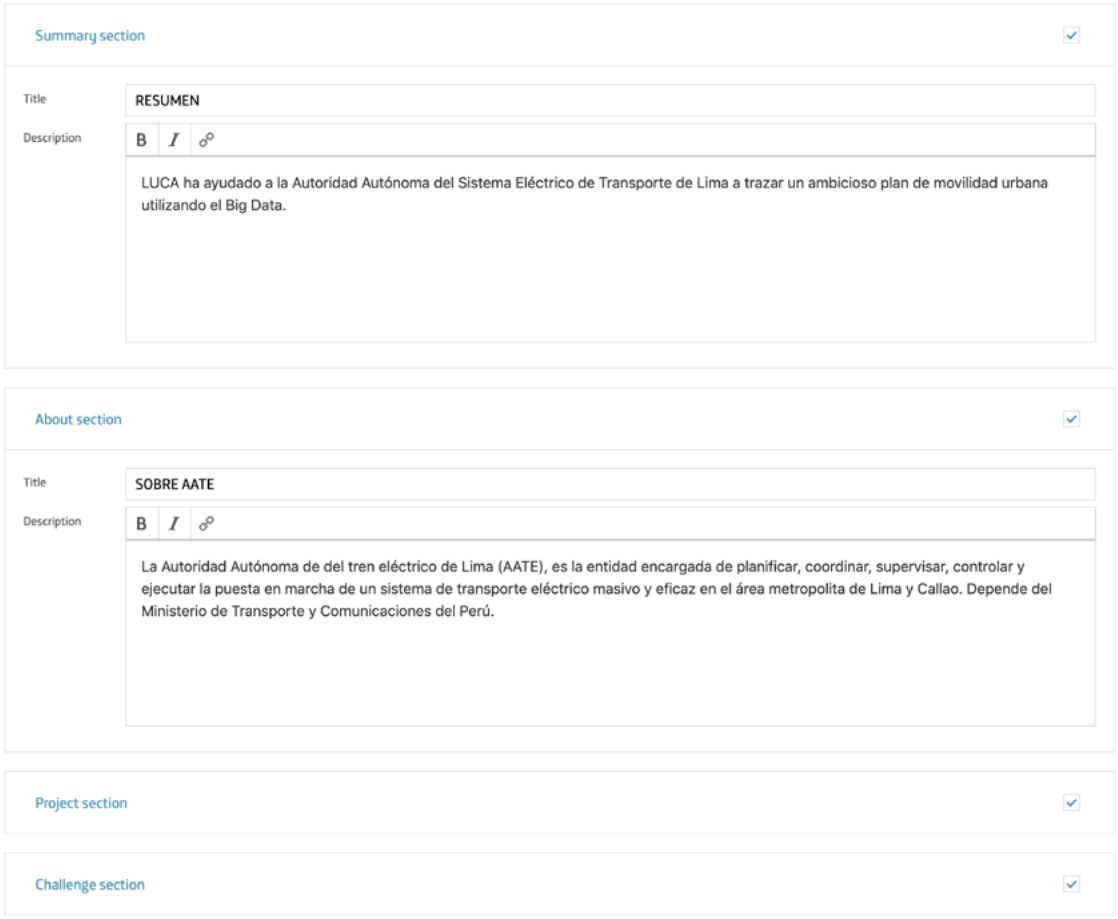
En este módulo se han desarrollado 14 submódulos, cada uno de ellos con una complejidad diferente. De hecho, la complejidad algunos de los submódulos puede llegar a ser mayor que la de un módulo simple. Cabe destacar dos tipos de submódulos: uno cuyos atributos son conocidos y previamente definidos y otro que puede tener un número de datos variable. Esta característica provoca mucha complejidad en cuanto a su gestión se refiere, ya que estos datos son almacenados en listas de submódulos y su representación visual es muy difícil de gestionar. Para solucionar este problema se ha desarrollado un componente *front-end* en JavaScript que permite añadir, eliminar y ordenar los elementos de la lista

.

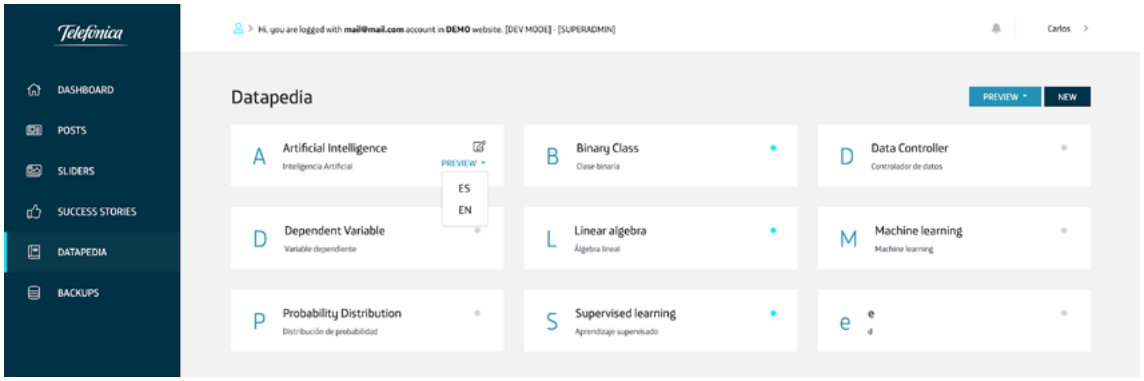
Cada submódulo implementa su propio modelo de datos, pero todos ellos contienen un atributo show de tipo *booleano* que indica si dicho submódulo se añade a la página de detalle o no. A continuación se muestra el modelo de datos del submódulo *WhyModule* y su validación.

```
public class WhyModule {

    private Boolean show;
    private Image image;
    private Language<String> title;
    private Language<String> desc;
    private Link link;

    public List<ValidationError> validate(String name) {

        final List<ValidationError> errors = new ArrayList<>();

        if(show != null && show){

            if (image.validate(name + ".image") != null)
                errors.addAll(image.validate(name + ".image"));

            if (title.validate(name + ".title") != null)
                errors.addAll(title.validate(name + ".title"));

            if (desc.validate(name + ".desc") != null)
                errors.addAll(desc.validate(name + ".desc"));

            if (link.validate(name + ".link") != null)
                errors.addAll(link.validate(name + ".link"));
        }

        return errors;
    }
}
```

La inserción de los datos de los submódulos se lleva a cabo durante la creación o modificación del módulo. Para que aparezcan los submódulos en la ventana de edición será necesario que la opción *detail* esté marcada. Al marcar esta opción, el campo *link* del módulo pasa a ser un campo obligatorio, y además debe ser único. Por lo tanto al guardar un caso de éxito, se comprueba que la dirección de su página de detalle asociada no exista en alguno de los módulos de *Success Stories* existentes en la base de datos. En caso contrario, se indica al usuario que esa dirección ya está en uso.



Para los módulos complejos se ha desarrollado la posibilidad de previsualizar su contenido aunque todavía no se haya publicado. De esta manera es posible previsualizar tanto la página general como la de detalle para asegurar que el contenido se representa correctamente en la vista pública. Se muestra un ejemplo de esto para el módulo de *Datapedia*.



El módulo *Datapedia* es muy similar, pero su submódulo *Page* introduce un concepto diferente a la hora de introducir los datos. Este módulo es un diccionario tecnológico y la página de detalle que puede llevar asociada no está formada por secciones, sino por texto plano con diferentes formatos de texto (encabezado, titulares, listas, etc.) e imágenes. Como la estructura de este texto es muy variable entre unas entradas del diccionario y otras, en lugar de utilizar un mecanismo de edición mediante submódulos, se ha utilizado un editor de texto enriquecido TinyMCE[2] para que el usuario introduzca este texto de forma libre, pero de manera que solo pueda utilizar las herramientas de estilo que se han especificado para este tipo de páginas. A continuación se muestra el código JavaScript con la configuración del editor que se ha utilizado y una figura con el editor integrado en una página de edición.

```
tinymce.init({

        selector: '.extendedTinyMCE textarea',
        height : "480",
        formats: {
            bold: {inline : 'span', 'classes' : 'bold'},
            italic: {inline : 'span', 'classes' : 'italic'}
        },
        inline_styles : true,
        menubar:false,
        plugins: "lists link paste image",
        toolbar: "styleselect | bullist | bold italic | link image",
        paste_data_images: false,
        force_br_newlines : true,
        force_p_newlines : true,
        forced_root_block : '',
        paste_as_text: true,
        style_formats: [
            {title: 'Header 1', format: 'h1'},
            {title: 'Header 2', format: 'h2'},
            {title: 'Header 3', format: 'h3'},
            {title: 'Header 4', format: 'h4'},
            {title: 'Header 5', format: 'h5'},
            {title: 'Header 6', format: 'h6'},
            {title: 'Paragraph', format: 'p'}
        ],
        content_css: [
            '../../../public/min/css/dashboard/tinyMCE.min.css'
        ]

});
```

2 Editor de texto enriquecido TinyMCE: https://www.tiny.cloud/

Formats

Header 1

Header 2

Header 3

Header 4

Header 5

Header 6

Paragraph

tiva

ias y modelos de datos pasados para intentar predecir
onar en el futuro

e a la pregunta ¿Qué puede pasar en el futuro según si nos basamos en lo que ha

Analiza las tendencias y modelos de datos pasados para intentar predecir cómo van a evolucionar en el futuro. Por ejemplo, una empresa puede predecir el crecimiento del negocio extrapolando el comportamiento en el pasado y asumiendo que no haya cambios relevantes en el entorno. La analítica predictiva facilita mejores recomendaciones y respuestas a preguntas a las que no puede atender el BI.

Aplicaciones de la Analítica Predictiva en diferentes entornos y sectores

En entornos financieros para asignar a los clientes un "credit score" o valor que predice la probabilidad de que ese cliente pague sus facturas puntualmente. También lo usan las grandes empresas de retail para identificar patrones de compra en los clientes, hacer predicciones de inventario o de los productos que suelen comprarse juntos (para ofrecer recomendaciones personalizadas) etc.

- En entornos financieros para asignar a los clientes un "credit score" o valor que predice la probabilidad de que ese cliente pague sus facturas puntualmente.
- También lo usan las grandes empresas de retail para identificar, hacer predicciones de inventario o de los productos que suelen comprarse juntos (para ofrecer recomendaciones personalizadas).
- También lo usan las grandes empresas de retail para identificar, hacer predicciones de inventario o de los productos que suelen comprarse juntos (para ofrecer recomendaciones personalizadas).

H1

POWERED BY TINY

5.3. Crear un módulo nuevo

Desde el inicio del desarrollo de este proyecto se ha diseñado la arquitectura del sistema de forma que se facilite el desarrollo de nuevos componentes. Al ser un gestor de contenidos basado en módulos es muy importante que el desarrollo e integración de nuevos módulos se realicen de la manera más sencilla y rápida posible. Aún así, crear un módulo nuevo, sobre todo si es complejo, no es trivial y puede llevar entre horas y semanas de trabajo dependiendo de la complejidad y el volúmen de datos que se quieran gestionar.

En este apartado se pretende explicar a alto nivel, pero de forma técnica, cuáles son los pasos a seguir para desarrollar un módulo nuevo.

- En primer lugar, se deben conocer todos los datos que se desean gestionar con el nuevo módulo y determinar si es de tipo simple o complejo según lo especificado en este capítulo.

- En segundo lugar, se debe implementar el modelo de datos con todos los atributos y el *DAO* del módulo indicando el nombre de la colección de la base de datos. La validación de los datos en el modelo podría hacerse en este momento, pero se recomienda hacer al final del todo, una vez que esté en funcionamiento.

- En tercer lugar es necesario crear las vistas de gestión (listar, crear, actualizar, etc) con plantillas Twirl. Esta tarea es muy sencilla si se utilizan las plantillas que se han desarrollado previamente y los ayudantes de vista que se han implementado para este proyecto. También será necesario implementar en Twirl las vistas de la parte pública donde los usuarios visualizarán la información.

- En cuarto lugar, se debe implementar el controlador asíncrono que se encargue de gestionar todas las peticiones de listado, creación, edición y borrado. Para la gestión de peticiones desde la parte pública de la web se necesita otro controlador más sencillo que recupere los datos del modelo y los conecte con una vista.

- En quinto lugar se tienen que definir en el archivo de *routes* la asociación entre los controladores y las *URLs* que se quieran utilizar.

La transformación de los datos del modelo a documentos de la base de datos se hace de forma automática gracias a que se ha conectado el *framework* con el driver asíncrono de MongoDB utilizando las funciones que éste define para los modelos de datos tipo *POJO*.

6. MÓDULOS DE ADMINISTRACIÓN

En cualquier gestor de contenidos, además de módulos relacionados con la gestión de contenidos e información propiamente dichas, se necesitan módulos enfocados a otros objetivos. En este proyecto se han implementado dos módulos cuya funcionalidad no está relacionada de forma directa con los contenidos y su visualización.

Durante el desarrollo del proyecto han ido surgiendo necesidades que se han resuelto implementando módulos que proporcionen soluciones para dichas necesidades. Uno de estos módulos se ha desarrollado para facilitar al usuario datos del tráfico del portal web, en particular: conocer las páginas más visitadas, los dispositivos desde los que se accede, etc. Por otra parte, se ha implementado un módulo para cubrir la necesidad de gestionar copias de seguridad de la base de datos.

6.1. Módulo de analítica y recopilación de datos

Este gestor de contenidos se está desarrollando para que equipos de *Marketing* puedan gestionar, de forma autónoma, determinadas secciones de un sitio web sin necesitar un perfil técnico. Una de las necesidades que estos equipos han planteado es la de realizar un seguimiento de las métricas de visualización de la parte pública del portal web.

Los datos recopilados se han obtenido de Google Analytics[1], una herramienta de analítica web que obtiene todos los datos relacionados con el tráfico que llega a una página web. Permite medir la audiencia de forma cuantitativa y cualitativa, el tiempo que permanece cada usuario en cada página, los dispositivos y navegadores mediante los cuales accede, el país desde el cual se accede, etc.

1 Google Analytics: https://analytics.google.com/analytics/web/

La obtención de los datos se lleva a cabo en tiempo real haciendo uso de la *API* que ofrece Google Analytics y para la representación visual de estos datos se hace uso de Chart.js[2], una librería JavaScript de código abierto para la visualización de datos en forma de gráficas.

En el momento de redactar esta memoria, los datos que se muestran en las gráficas corresponden a las métricas del sitio web al que va a sustituir este gestor de contenidos. Una vez se despliegue este proyecto en producción, comenzarán a aparecer los datos correspondientes a este último.

Las métricas obtenidas se han representado en un tablón (*dashboard*) que funciona como página principal del gestor de contenidos. En este tablón se han organizado los datos en tres secciones que a su vez muestran diferentes métricas y comparaciones.

Duración y número de visitas (en los últimos 30 días)

- Número de sesiones[3] por usuario
- Duración de la sesión por usuario
- Número de páginas visitadas por sesión

Páginas más visitadas y visitas por países (en los últimos 30 días)

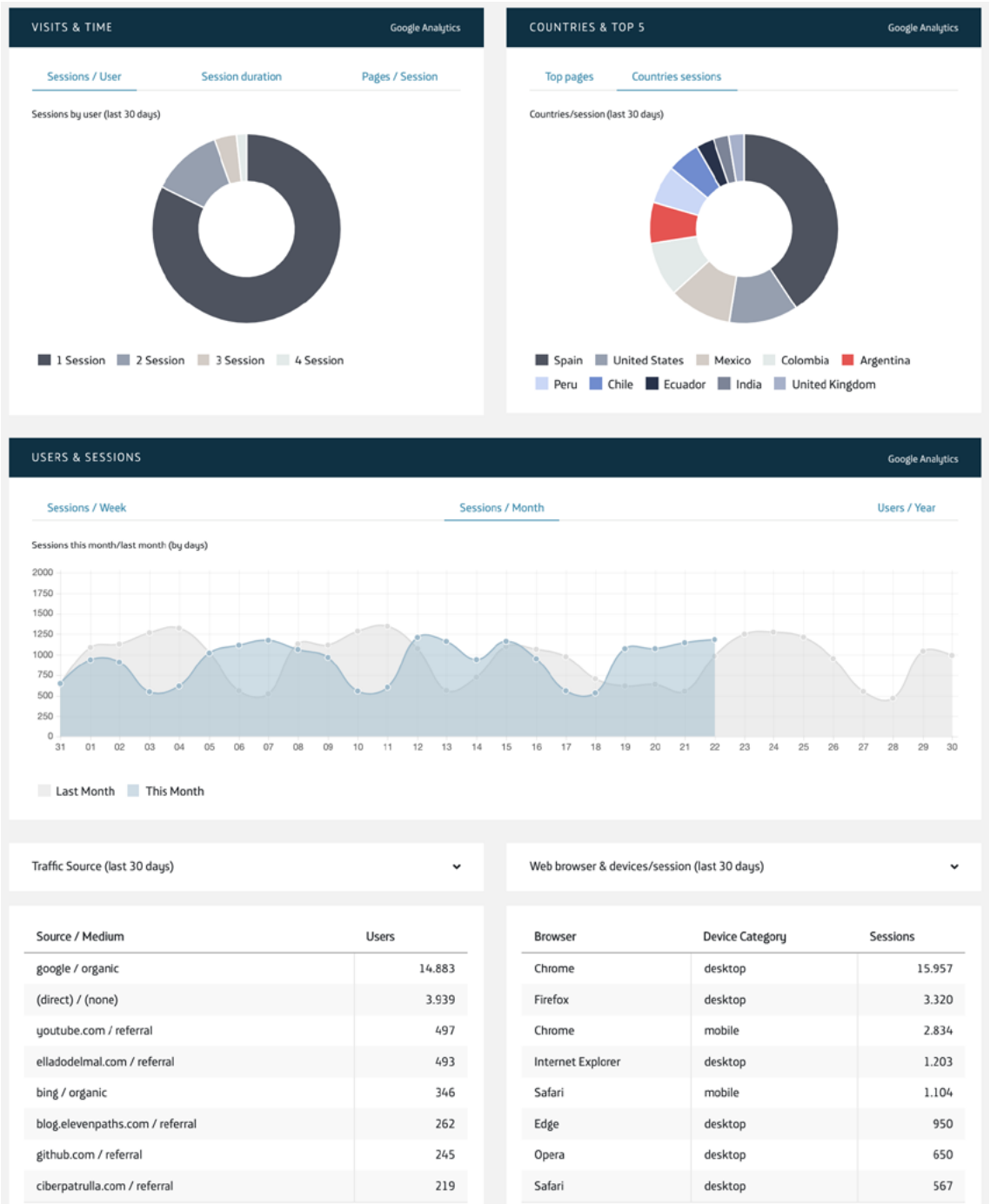
- Listado con las 5 páginas más visitadas
- Países con mayor número de visitas

Usuarios y sesiones

- Comparativa del número de sesiones de la semana actual con el de la semana pasada
- Comparativa del número de sesiones del mes actual con el del mes pasado
- Comparativa del número de usuarios de este año con el del año pasado

2 Chart.js: https://www.chartjs.org/

3 Una sesión es una visita a la página web. Por ejemplo, un usuario que realice tres visitas contabilizará como tres sesiones y un solo usuario.



Source / Medium	Users
google / organic	14.883
(direct) / (none)	3.939
youtube.com / referral	497
elladodelmal.com / referral	493
bing / organic	346
blog.elevenpaths.com / referral	262
github.com / referral	245
ciberpatrulla.com / referral	219

Browser	Device Category	Sessions
Chrome	desktop	15.957
Firefox	desktop	3.320
Chrome	mobile	2.834
Internet Explorer	desktop	1.203
Safari	mobile	1.104
Edge	desktop	950
Opera	desktop	650
Safari	desktop	567

Cada consulta se realiza ejecutando una consulta a través de una petición *AJAX* a la *API* de Google Analytics, el cual devuelve los datos para representarlos en forma de gráfica con Chart.js. A continuación se muestran fragmentos de código que indican cómo se realizan las peticiones.

```
// Top 5 páginas más visitadas en los últimos 30 días
function renderTopPageviewsSessionChart() {

  // Query con los parámetros de las métricas
  query({
    'ids': 'ga:73270755',
    'dimensions': 'ga:pageDepth',
    'metrics': 'ga:sessions',
    'sort': '-ga:sessions',
    'max-results': 5,
    'start-date': '30daysAgo',
    'end-date': 'yesterday'
  })

    .then(function(response) {

      var data = [];
      var colors = ['#4D5360', '#949FB1', '#D4CCC5', '#E2EAE9'];

      response.rows.forEach(function(row, i) {
        data.push({
          label: row[0] + ' Pages',
          value: +row[1],
          color: colors[i]
        });
      });
      // Construcción de la gráfica con Chart.js
      new Chart(makeCanvas('chart-container-pageviews-session'))
            .Doughnut(data);
      generateLegend('legend-container-pageviews-session', data);
    });

}


// Función que ejecuta la petición a la API
function query(params) {
  return new Promise(function(resolve, reject) {

    var data = new gapi.analytics.report.Data({
      query: params
    });
    data.once('success', function(response) {
      resolve(response);
    })
    .once('error', function(response) {
      reject(response);
    })
    .execute();
  });

}
```

6.2. Módulo de copias de seguridad automáticas

El módulo de copias de seguridad se ha implementado para gestionar la información que se almacena en la base de datos MongoDB. Gracias a este módulo es posible mantener una copia de seguridad con el estado de la información del gestor de contenidos en un momento concreto.

El intervalo de tiempo con el que se realizan las copias de seguridad se define en el archivo de configuración. En este caso, el valor por defecto es de 24 horas. El momento del día en el que se desea realizar la copia de seguridad se especifica mediante una expresión de tipo CRON[4]. La primera vez que se arranca la aplicación web se calcula el tiempo restante para realizar la próxima copia de seguridad, parámetro que se introduce como *Delay* (tiempo de demora). A partir de este momento, cada 24 horas, se hace una llamada al método *newBackup*() del controlador *Backup*.

Al inicio del proyecto las copias de seguridad se realizaban ejecutando un comando de MongoDB mediante el método *exec*() de Java, el cual permite la ejecución de comandos arbitrarios en el sistema operativo. Por motivos de seguridad, la aplicación web no puede tener permisos para ejecutar este tipo de comandos, y por esta razón se ha desarrollado un sistema de copias de seguridad desde cero. Como todas las colecciones relacionadas con el gestor de contenidos se encuentran dentro de una misma base de datos, la forma más adecuada de hacer una copia de los datos es recorrerlos de forma manual. El funcionamiento de este sistema es el siguiente:

- Primero se listan todas las colecciones de la base de datos
- Para cada una de las colecciones, se itera, de manera asíncrona, sobre todos los documentos que contiene, se convierten a formato JSON y se van añadiendo a una lista.
- Cada lista, es decir, cada colección, se guarda en un fichero con extensión *.json* y se envía al servidor de Amazon S3.

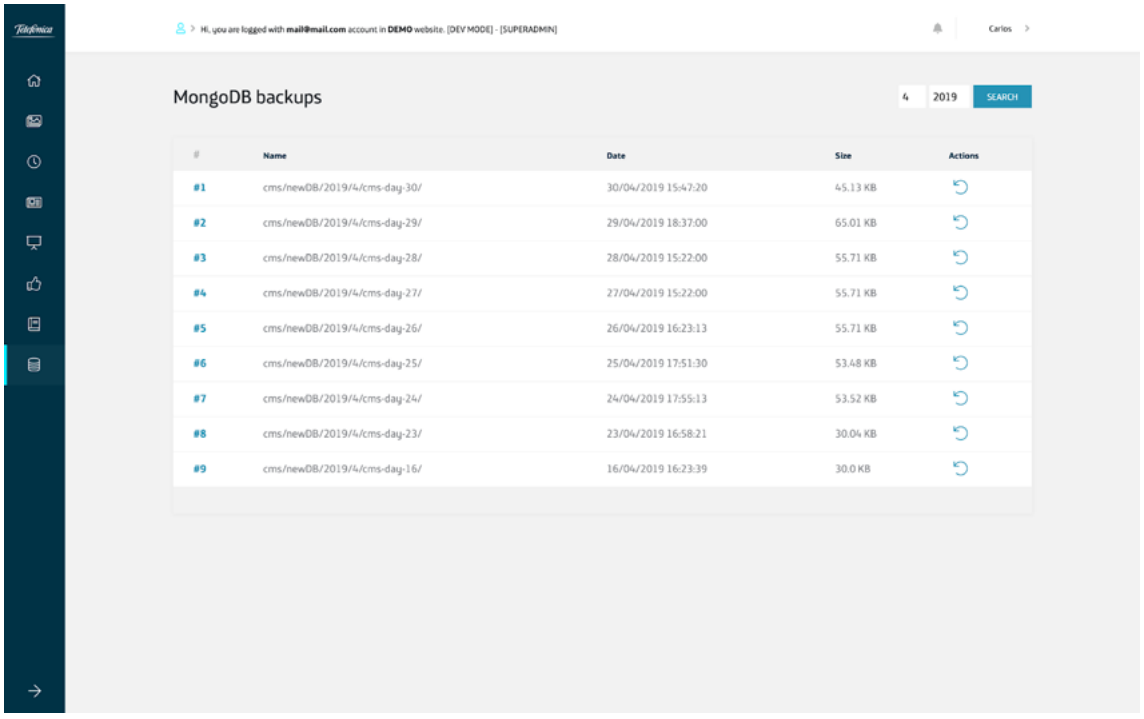
De esta forma, se mantiene cada copia de seguridad en un directorio independiente cuyo nombre sigue la estructura "*cms/newDB/2019/5/cms-day-23/*". Dentro de cada carpeta se encuentran tantos ficheros JSON como colecciones existan en la base de datos.

4 CRON expressions: https://docs.oracle.com/cd/E12058_01/doc/doc.1014/e12030/cron_expressions.htm

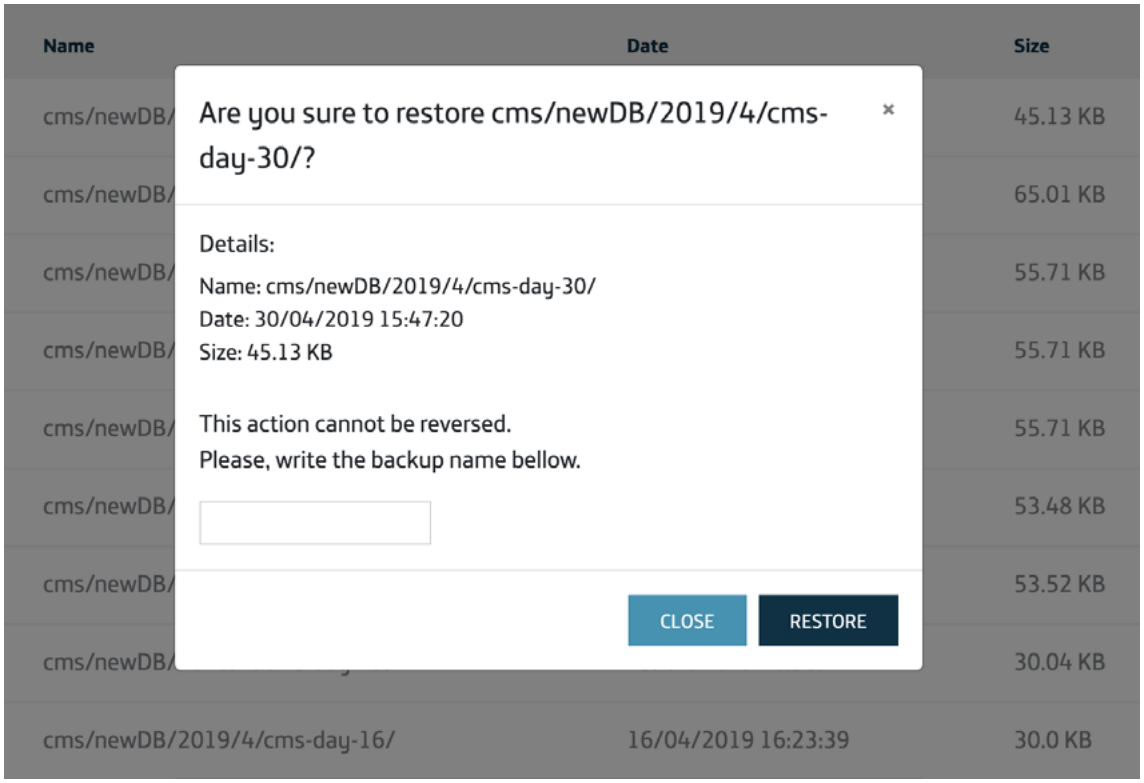
De la misma forma que se proporciona un sistema para hacer copias de seguridad, se ha implementado la operación complementaria: la restauración. La posibilidad de restaurar una copia de la base de datos estará restringida al rol de usuario más alto (*superadmin*).

El funcionamiento de la restauración se asemeja mucho al de las copias. Se itera sobre todos los archivos *JSON* que contiene la carpeta de la copia de seguridad que se desea restaurar. Para cada uno de ellos se elimina la colección existente en la base de datos, después se recorre cada documento de cada archivo y se inserta en la colección.

El módulo dispone de una interfaz visual que facilita la tarea de restaurar una copia en concreto. Se muestran, en forma de lista, todas las copias de seguridad disponibles en el mes actual junto a la fecha en la que se realizaron, el tamaño que ocupa cada copia y un botón que realiza la restauración. En esa misma página, mediante un pequeño formulario, se ofrece la posibilidad de navegar por todos los meses y años posibles por si se necesitase restaurar una copia de seguridad más antigua. Este formulario se valida con expresiones regulares tanto en el *front-end* como en el *back-end* y realiza las peticiones mediante una *URL* paramétrica.



Para confirmar que la acción de restaurar se esté haciendo de manera plenamente consciente se requiere, mediante una ventana modal de confirmación, que se introduzca la ruta completa donde se encuentra la copia. De esta forma resulta difícil llevar a cabo una restauración de manera accidental.



7. CONCLUSIONES Y TRABAJO FUTURO

En esta última sección se concluye detallando el cumplimiento de los objetivos iniciales, los problemas que han surgido durante el desarrollo del proyecto y las características que en un futuro cercano se implementarán para añadir más funcionalidad al gestor de contenidos.

7.1. Objetivos alcanzados

El cumplimiento de los objetivos marcados al inicio del proyecto se ha llevado a cabo en su totalidad. Además, se han incluido características nuevas que no se marcaron como requisitos, como la integración con una API privada que elimina los metadatos de los ficheros antes de subirlos a Amazon S3 o la conexión con las *APIs* de Twitter, YouTube y Wordpress para obtener las últimas publicaciones.

El objetivo principal de este proyecto ha sido, desde un principio, el desarrollo de un gestor de contenidos que permita gestionar secciones de una web sin necesidad de tener un conocimiento técnico. Este objetivo se ha alcanzado de manera satisfactoria, ya que a día de hoy el gestor de contenidos está desplegado en un entorno de pruebas siendo utilizado por varios usuarios de forma simultánea para probar su correcto funcionamiento.

Los módulos de gestión de publicaciones y *sliders*, de casos de éxito y el diccionario tecnológico están totalmente terminados y en funcionamiento. El portal de analítica de datos muestra las métricas especificadas en el capítulo 6, y el módulo de copias de seguridad funciona con normalidad. De hecho, durante el desarrollo, este último módulo ha sido imprescindible para no perder tiempo en introducir manualmente todos los datos de nuevo, sino recuperarlos de forma automática cuando se dañan o pierden durante el desarrollo.

7.2. Dificultades encontradas

Durante las distintas fases del desarrollo del proyecto se han encontrado muchas dificultades que se han ido resolviendo de forma progresiva investigando tecnologías y aprendiendo de manera simultánea al desarrollo del gestor. Algunas de ellas se enumeran a continuación:

- La primera dificultad viene dada por el hecho de haber usado la última versión del *framework* Play, una versión totalmente nueva que ha reescrito completamente el *framework* y la forma en la que funciona. Gracias a la documentación oficial se han conseguido superar todas las barreras que han ido surgiendo.

- La segunda dificultad vino dada por la necesidad de comprender el concepto de asincronía en una aplicación web e implementarla aprendiendo a utilizar los nuevos mecanismos de asincronía que incorpora Java en su versión 8, para conectar de manera correcta el *framework* con el sistema gestor de base de datos.

- Otro de los problemas ha sido el de diseñar la arquitectura para que la asociación de los modelos de datos con los atributos de la base de datos fuese automática (mapeado). Este problema se ha resuelto utilizando el adaptador de MongoDB para clases *POJO*.

- La conexión con las *APIs* externas y el desarrollo de una propia para la carga de ficheros ha supuesto un avance importante de conocimiento, pero aprender cómo funcionan estos sistemas ha requerido de mucho tiempo de formación extra.

7.3. Trabajo futuro

En este punto del desarrollo podemos afirmar que el proyecto es viable para cumplir la función de un gestor de contenidos y cubrir las necesidades de la empresa. No obstante, hay un gran número de características que se desean implementar en un futuro cercano. Algunas de ellas se describen a continuación.

- Módulo de formularios. Implementar un módulo que permita gestionar formularios de contacto e inscripción a eventos desde el panel de administración del gestor de contenidos.

- Módulo de archivos. Se ha implementado un módulo que permite subir ficheros sin necesidad de que estén asociados a un modelo de datos. Se plantea ampliar su funcionalidad para permitir acciones como borrar ficheros, eliminar aquellos que no estén referenciados, buscar duplicados, etc.

- Módulo de SEO. Implementar un módulo que permita editar la configuración del SEO de la página desde una interfaz visual, en lugar de hacerlo desde el código.

- Módulo de copias de seguridad de ficheros. Implementar un módulo que realice copias de seguridad de los archivos subidos al servidor de Amazon S3 en otro servidor independiente.

8.1. CONCLUSIONS AND FUTURE WORK

This last section concludes detailing to which extent the initial objectives have been met, the problems that have arisen during the development of the project and the features that will be implemented in the near future to add more functionality to the content manager.

8.1. Achieved goals

The fulfillment of the objectives defined at the beginning of the project has been completely carried out. In addition, new features, that were not marked as requirements, have been included, such as integration with a private API that removes the metadata from the files before uploading them to Amazon S3 or the connection with the APIs of Twitter, YouTube and Wordpress to obtain the latest publications.

The main objective of this project has been, from the beginning, the development of a content manager that allows one to manage sections of a website without the need of having technical knowledge. This objective has been achieved satisfactorily, since today the content manager is deployed in a test environment and it is being used by several users simultaneously to test its proper functioning.

The modules of management of publications and sliders, Success Stories and the technological dictionary are completely finished and in operation. The data analytics portal shows the metrics specified in chapter 6, and the backup module works normally. In fact, during the development, this last module has been essential in order not to waste time re-introducing all the data manually when they are damaged or lost by allowing to recover them automatically.

8.2. Difficulties encountered

During the different phases of the project development, many difficulties have arisen. These have been progressively resolved by investigating technologies and learning simultaneously to the development of the manager. Some of them are listed below.

- The first difficulty was due to having used the latest version of the Play framework, a totally new version that has completely re-written the framework and the way it works. Thanks to the official documentation, all the issues that have appeared have been overcome.

- The second difficulty was due to the need to understand the concept of asynchrony in a web application and to implement it by learning to use the new asynchrony mechanisms that Java incorporates in its 8th version, in order to correctly connect the framework with the database management system.

- Another problem has been the design of the architecture so that the association of the data models with the attributes of the database could be automatic (mapping). This problem has been solved by using the MongoDB adapter for POJO classes.

- The connection with the external APIs and the development of an own API for loading files has meant an important advance of knowledge. Nevertheless, learning how these systems work has required a lot of extra training time.

8.3. Future work

At this point of the development we can claim that the project is feasible to fulfill the function of a content manager and cover the needs of the company. However, there are a large number of features that could be implemented in the near future. Some of them are described below.

- Form management module. Implementing a module that allows to manage contact and registration to events forms from the administration panel of the content manager.

- File module. A module has been implemented that allows uploading files without the need to be associated with a data model. It is proposed to extend its functionality to allow actions such as deleting files, eliminating those that are not referenced, searching for duplicates, etc.

- SEO module. Implementing a module that allows editing the SEO settings of the page from a visual interface, instead of doing it in the code.

- File backup module. Implementing a module that backs up the files uploaded to the Amazon S3 server on another independent server.

BIBLIOGRAFÍA

[1] Foy Julien. *Play framework essentials: an intuitive guide to creating easy-to-build scalable web applications using the Play framework*, Birmingham: Packt Publishing, 9781783982400, 2014.

[2] Nicolas Leroux and Sietse de Kaper. *Play for Java*. Shelter Island, 9781617290909, 2014.

[3] Vohra, D. *Pro Mongodb Development; The Expert's Voice in Databases*; Apress: New York, New York, 2015.

[4] Spell, B. *Pro Java 8 Programming; The Expert's Voice in Java*; Apress: Berkeley, CA, 2015.

[5] Sarcar, V. *Java Design Patterns : A Tour of 23 Gang of Four Design Patterns in Java*; The Expert's Voice in Java; Apress: Berkeley, CA, 2016.

[6] Gulabani, S. *Practical Amazon Ec2, Sqs, Kinesis, and S3 : A Hands-On Approach to Aws*; Itpro Collection; Apress: New York, NY, 2017.

[7] Shah, N.; Balda, G. *Html5 Enterprise Application Development*; Packt Pub: Birmingham, 2013.

[8] Frain, B. *Sass and Compass for Designers*; Packt Publishing, 2013.

[9] Goodman, D. *Javascript Bible*, 7th ed.; Wiley: Hoboken, N.J., 2010.

[10] Chaffer, J.; Swedberg, K. *Learning Jquery*, 4th ed.; Community Experience Distilled; Packt Publishing: Birmingham, 2013.